Escola Politècnica Superior de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL: Estudi i simulació d'algorismes d'encaminament amb múltiples restriccions
AUTOR: Guillem Torrens Marín

DIRECTORES: Anna Agustí Torra, Cristina Cervelló Pastor

DATA: 24 de febrer de 2005

Títol: Estudi i simulació d'algorismes d'encaminament amb múltiples restriccions

Autor: Guillem Torrens Marín

Directores: Anna Agustí Torra, Cristina Cervelló Pastor

Data: 24 de febrer de 2005

Resum

El problema que es planteja en aquest projecte està ambientat en el transport ferroviari. Concretament ens centrarem en el tràfic dins de les estacions. A cada estació hi ha un CCT (Centre de Control de Tràfic) des d'on es visualitza i controla el moviment dels trens.

En una estació hi ha diferents objectius per complir. El més important és la seguretat, ja que no es pot deixar de complir en cap cas. Un altre objectiu molt important és l'eficiència. Els trens s'han de desplaçar dins de l'estació gastant el mínim temps possible. Aquest projecte té com a missió optimitzar l'eficiència.

Durant el recorregut del tren a l'estació, aquest s'aturarà en algun estacionament o si troba un senyal en vermell. El CCT reserva els circuits de via per on ha de circular el tren, i posa les agulles en la posició correcta.

Actualment, la persona encarregada del CCT selecciona manualment l'itinerari que ha de seguir el tren. Això pot causar molts errors i problemes de seguretat, així com pèrdua de temps.

Per això es fa un paral·lelisme entre l'estació de tren i una xarxa de comunicacions, on l'estació és un graf i els circuits de via són els nodes. I com si fos una xarxa, s'aplica un algorisme d'encaminament per a cercar el camí més curt entre dos punts.

Una condició pròpia dels trens, un cop estan seguint un camí, és que no poden tornar enrere en aquella mateixa ruta. És a dir, que els trens poden circular per totes les vies en els dos sentits, però no poden circular en els dos sentits en un mateix trajecte. Això planteja una restricció.

Per a implementar l'aplicació és necessari utilitzar un llenguatge de programació. S'ha escollit un llenguatge orientat a objectes, pels avantatges que proporciona. *Java* és la plataforma ideal per complir els objectius.

Title: **Estudi i simulació d'algorismes d'encaminament amb múltiples Restriccions**

Author: Guillem Torrens Marín

Directors: Anna Agustí Torra, Cristina Cervelló Pastor

Date: Fabruary, 24th 2005

Overview

The main project of this research is the railway environment, which is particularly focused on the traffic within railway facilities. This one is controlled and recorded by the TCC (Traffic Control Center).

In regards of train traffic, there are two main standards to be achieved. The first one is security, which must be mandatory at all times. Close related to this one, there is the efficiency. At the time that trains follow security procedures, they have to reach each of the stations using the minimum time possible. This project wants to improve efficiency.

Trains run along the railways either stopping at train's parking or at red traffic lights. TCC is in charge of scheduling traffic within rail circuits and setting all correct signals for that purpose.

Nowadays, these mentioned duties are all carried out manually by TCC staff. This normal procedure causes human mistakes that mainly cause waste of time and failure at security policies.

This project shows a parallelism between the train traffic and a communication network, where the train station is a graph and the rail circuits are edges. An algorithm is used to find the shortest way between two given points as done in a network.

The rule to follow is that trains can run in both directions on the railways. But once they follow a destination, trains can not go forward and back. This restricts use of trains causes a constraint.

To implement the application, a programming language is required. The one selected in this research is an object oriented language, which provides with lots of advantages. The mentioned framework is Java.

ÍNDEX

INTRODUCCIÓ 1

CAPÍTOL 1. RAONS DEL PROJECTE 3

1.1. Finalitat i condicions 3

1.2. Mètodes utilitzats 4

1.3. Objectius 4

1.4. Planificació 5
1.4.1. Descripció de les divisions de treball 5
1.4.2. Diagrama temporal 7

1.5. Glossari 8

CAPÍTOL 2. METODOLOGIA 11

2.1. Algorisme de Dijkstra amb restriccions 11
2.1.1. Raons d'utilitzar *Dijkstra* 12
2.1.2. Restriccions físiques i restriccions lògiques 13

2.2. El llenguatge de programació Java 13
2.2.1. Avantatges d'utilitzar *Java* 14
2.2.2. *Swing* i AWT per a fer GUIs 15
2.2.3. UML, la notació 15

CAPÍTOL 3. CONTACTE AMB *JAVA* 17

3.1. Programari necessari per al desenvolupament del projecte: J2SE de Java i l'SDK NetBeans 3.6 17

3.2. APIs i tutors 17
3.2.1. Realització de proves 18

CAPÍTOL 4. ESPECIFICACIÓ I DISSENY DE L'APLICACIÓ 19

4.1. L'aplicació 19
4.1.1. Entrada i sortida de l'aplicació 20
4.1.2. Interfície gràfica i línia de comandes 24
4.1.3. Captura d'excepcions (errors) 25

4.2. UML 26
4.2.1. Diagrama de classes 26
4.2.2. Llistat de les classes 28
4.2.3. Diagrama de seqüència 28

CAPÍTOL 5. DESENVOLUPAMENT I PROVES 31

5.1. Interfície gràfica 31

5.2. Línia de comandes 36

5.3. Captura d'excepcions 38
5.3.1. Errors de format en els fitxers 39
5.3.2. Errors de validesa en els fitxers 40
5.3.3. Altres errors 41

CAPÍTOL 6. VALORACIÓ 43

6.1. Assoliment dels objectius 43

6.2. Variacions respecte la planificació inicial 44

6.3. Solució a un cas real? 44

6.4. Ampliacions futures 45

6.5. Conclusions personals 46

CAPÍTOL 7. BIBLIOGRAFIA 47

ANNEX 1. DESCRIPCIÓ DE LES CLASSES 49

ANNEX 2. DESCRIPCIÓ DEL FITXER D'ENTRADA 59

ANNEX 3. DESCRIPCIÓ DEL FITXER DE SORTIDA 63

ANNEX 4. PLÀNOL DE L'ESTACIÓ DE SANTS 65

INTRODUCCIÓ

En aquesta introducció s'explica com està feta la divisió dels capítols.

En el primer capítol s'exposen les raons del projecte, la seva finalitat, les condicions necessàries per implementar-lo, i la manera com desenvolupar la feina. Hi ha hagut canvis al llarg del projecte respecte a la planificació inicial. Més endavant, s'expliquen aquests canvis i les raons per fer-los. Al final del capítol hi ha un glossari amb els termes més importants.

Abans de començar el projecte es van fer reunions per conèixer quins eren els mètodes que s'havien d'utilitzar durant el projecte. En el segon capítol s'expliquen aquests mètodes, que són l'algorisme de *Dijkstra* amb les modificacions que comporten les restriccions i l'arquitectura *Java*. Hi ha la definició de la metodologia de disseny d'aplicacions UML.

En el tercer capítol hi ha una ambientació de l'entorn de treball. S'explica al detall el programari necessari, les proves que es van fer per familiaritzar-se amb *Java*. La quantitat de documentació necessària era molt elevada, i es van utilitzar mecanismes de recerca per tal d'optimitzar el temps. Durant la implementació del programa s'ha de consultar la documentació de manera freqüent, per tant, és necessari agilitar el procés.

Al quart capítol es mostra l'especificació del disseny de l'aplicació, la manera com s'implementarà el programa. S'expliquen les línies seguides durant el desenvolupament. S'exposen dos diagrames, un de seqüència i un de classes, que són el resultat d'aplicar la metodologia UML.

El capítol cinquè se centra en el desenvolupament de l'aplicació, la seva implementació. El procés és el més important i el que va comportar més feina. S'explica detalladament el funcionament del programa. La seqüència de figures d'aquest capítol mostren l'ús de l'aplicació en les seves dues modalitats. Es detalla el tractament dels errors, juntament amb totes les proves fetes al llarg de la implementació.

En el capítol sisè es valora el projecte i s'extreuen unes conclusions, els quals són dos temes molt lligats. S'expliquen els canvis ocorreguts durant l'aplicació, així com la comparació de l'aplicació inicial demanada i de la obtinguda. Finalment es comprova si se soluciona el problema del cas real.

En el capítol setè es presenta la bibliografia utilitzada al llarg del projecte.

Ens els annexos s'aprofundeix en la descripció de les classes exposades en el capítol quart. També es mostra una part del fitxer real utilitzat durant la implementació. Per acabar hi ha el plànol de l'estació de Sants, que és la que forma part del fitxer.

CAPÍTOL 1. RAONS DEL PROJECTE

En el primer capítol s'exposen les raons del projecte, la seva finalitat, quines condicions han sigut necessàries per implementar-lo, i la manera com es volia desenvolupar la feina.

1.1. Finalitat i condicions

En el tràfic ferroviari s'han de garantir certs serveis. Un és la seguretat, de cap manera no hi pot haver cap accident ni cap incident. Ni per culpa d'errors humans, ni per culpa d'errors no humans. Un altre objectiu que s'ha de complir, és el de l'eficiència, tots els trens han de recórrer els camins en qualsevol sentit i han de circular pel millor camí possible, sense que es deixi de garantir la seguretat.

El millor camí possible no sempre és el més curt o el de menys cost. A vegades es volen crear rutes preferents per motius que no siguin l'eficiència. En aquest projecte es vol optimitzar l'eficiència, però l'aplicació està feta de tal manera que, es poden introduir altres paràmetres a l'hora de dissenyar l'itinerari.

S'ha de tenir en compte que cada estació de tren és diferent, i a més a més, pot tenir canvis en la seva estructura. Actualment, a les estacions de tren de la ciutat de Barcelona l'itinerari que segueixen els trens es confecciona de manera manual. Això pot comportar diversos errors: equivocar-se de camí; triar un camí bo, però no el millor; errar en la posició de les agulles; no introduir correctament els noms de qualsevol dels elements, i tots els possibles errors que existeixen. Tot això és lògic, ja que és una feina molt repetitiva.

Un mètode alternatiu és crear una aplicació que faci la búsqueda del camí en lloc de fer-ho una persona. Podem enumerar els avantatges: eficiència; no errors (un cop s'ha comprovat el seu funcionament); gasta poca memòria i el temps d'execució és baix, ja que l'aplicació no és complexa.

Un cop ja sabem quin procés volem, hem d'establir les condicions per tal que l'aplicació funcioni en tots els casos, com poden ser diferents estacions, la validesa del fitxer d'entrada i del fitxer de sortida,...

- Tot i que funcioni per a qualsevol estació, les dades que conformen l'estació han de tenir un format comú. En el cas d'un fitxer de text, s'ha de mantenir completament aquest format.

- En les mateixes condicions que el punt anterior, el camí resultant, també s'ha d'expressar en un format predeterminat. Ja sigui en forma de fitxer de text, com de sortida per pantalla.

1.2. Mètodes utilitzats

Els mètodes utilitzats formen també part de les condicions del projecte.

El llenguatge de programació seleccionat per portar a terme el projecte és *Java*. En el següent capítol es parlarà més detalladament dels avantatges i inconvenients, i de les possibilitats que ens ofereix aquesta plataforma. Tant per la interfície gràfica, com per la resta de codi, es poden utilitzar totes les utilitats de *Java*. L'elecció de cada classe o objecte, o qualsevol altre detall, depèn de l'aplicació.

Ja que el projecte és sobre encaminament, es necessita un algorisme de cerca del millor camí. Aquest algorisme és el de *Dijkstra*. A l'algorisme s'hi ha d'afegir una restricció, ja que els trens tenen una restricció física en qualsevol cas. No poden anar enrere en relació a l'itinerari que estan seguint.

Com ja s'ha comentat abans la seguretat té prioritat per damunt de l'eficiència. Per tant, l'aplicació no pot obligar un tren a fer un itinerari si la seguretat no està garantida. Llavors, en el cas que s'arribés a un cas semblant, el programa no tindria validesa. És a dir, l'aplicació està a un nivell jeràrquic inferior al de la seguretat, que significa que funcionarà només si la seguretat compleix els requisits corresponents.

El graf que descriu l'estació, així com el camí que resulti de la búsqueda, aniran en format fitxer de text, tal com s'ha dit en l'apartat anterior.

1.3. Objectius

Hi ha tres objectius bàsics en aquest projecte:

1) Desenvolupar un projecte per optimitzar el tràfic ferroviari en una estació de tren.
2) Adquirir coneixements sobre *Java*.
3) Proporcionar una aplicació per a un ús real en una empresa.

Per assolir aquests objectius es requereixen un passos intermedis. S'ha de conèixer el problema, s'ha de dissenyar una possible solució (es farà mitjançant la metodologia UML) i s'ha d'implementar aquesta solució. S'utilitzarà el programari i la consulta necessària. Es dividirà el projecte en unes tasques, com veurem en el següent apartat.

El primer objectiu és el més important, es desenvolupa un projecte a partir d'un problema. Es fa un paral·lelisme entre el sistema ferroviari i un sistema de comunicacions. S'utilitzen mecanismes d'aquest últim sistema per tal de solucionar les necessitats.

El segon objectiu resulta complementari. El fet d'utilitzar un llenguatge de programació és necessari. S'utilitza la tecnologia *Java* per diferents motius:

permet construir sistemes semblants a la realitat mitjançant abstracció; les classes creades es poden tornar a utilitzar en qualsevol altre programa i proporciona aplicacions modulars. A més a més, s'adquireixen coneixements per a futurs projectes.

El tercer objectiu és el resultat del projecte, és a dir, l'aplicació que es creï ha de funcionar, i a més a més ha de complir els requisits que es necessiten per a una aplicació que funcioni en un escenari real.

1.4. Planificació

El nombre d'hores total del projecte són 420. Aquestes hores s'engloben en 15 setmanes, que és el temps de duració d'un quadrimestre. Si fem una divisió, ens surten 28 hores per setmana o 5,5 hores per dia lectiu.

Es divideix la feina en 7 tasques. Cadascuna d'elles amb un nombre d'hores diferents. Algunes feines són independents les unes de les altres, i algunes d'elles necessiten que l'anterior s'hagi acabat per poder-les començar. Això s'explica a continuació.

1.4.1. Descripció de les divisions de treball

1.4.1.1. Introducció al problema (28hores)

Primeres reunions amb la directora del treball. Es planteja el problema, les condicions, i el resultat que es desitja.
Hi ha un procés de lectura de la documentació. També hi ha un recerca d'informació, sobretot per Internet.

1.4.1.2. Descripció de les tasques a realitzar. (28hores)

Un cop es coneixen les dimensions del projecte es fan les divisions corresponents a cada part del treball. S'utilitza com a base el nombre de setmanes i hores disponibles.

El nombre d'hores que apareixen al final de cada tasca, són les hores pràctiques que s'ha trigat en realitzar cadascuna de les feines.

A cada apartat s'expliquen els passos que s'han seguit pel desenvolupament del treball.

1.4.1.3. Instal·lació del programari necessari. (56hores)

En aquesta tasca s'instal·la el programari necessari per a realitzar les aplicacions basades en Java. Es descarreguen els programes des de la pàgina oficial de Sun.
Dins de la mateixa pàgina hi ha la documentació, els exemples i fins i tot els tutors necessaris per a implementar tot tipus d'aplicacions. La documentació de les classes propietàries de Java caldrà tenir-les disponibles durant l'elaboració del codi.

1.4.1.4. Especificacions i disseny (56hores)

Aquesta tasca és molt important per a no perdre temps en la implementació de l'aplicació. Si és té molta cura en aquest procés, no s'haurà de repetir cap part del codi; s'evita haver de començar de nou en qualsevol punt si ens adonem que ens trobem en un camí sense sortida.

Per altra banda hi ha dos diagrames basats en la metodologia UML els quals ens mostren gràficament part del disseny. Cadascun d'ells té un significat diferent, i junts, es complementen i donen una visió global del projecte.

1.4.1.5. Implementació del programa i proves. (168hores)

En aquesta feina és indispensable tenir el disseny acabat. El resultat d'aquesta tasca és l'aplicació en funcionament i totes les proves fetes. Sempre hi ha millores o ampliacions per fer, i una condició que el programa ha de complir, és la capacitat d'executar-les en qualsevol moment.

Les proves necessàries es fan setmanalment per a trobar possibles errors. Hi ha proves de tot tipus: buscar problemes en el fitxer d'entrada mitjançant errors provocats, provar mètodes separadament del programa per tal de simplificar el treball, fer accions repetitives per veure si hi ha variables mal inicialitzades o altres valors erronis, etc.

1.4.1.6. Comprovació de la planificació inicial. Redacció. (56hores)

Es redacta tota la informació obtinguda i s'afegeix l'experiència obtinguda després del projecte, com els possibles problemes i solucions que s'hagin trobat. Es compara la planificació que s'havia fet al principi, amb el resultat final obtingut.

La redacció i el projecte s'elaboren simultàniament perquè quedi constància de tota la informació, com: les experiències, els errors i les seves solucions, els

canvis que hagin ocorregut, els seus resultats,... També serveix com a consulta dels desencerts que s'han fet i no tornar-los a repetir.

1.4.1.7. Preparació de la presentació. (28hores)

Aquesta última feina serveix per a preparar les transparències per la presentació oral del TFC. També cal elaborar un guió per exposar correctament la presentació.

1.4.2. Diagrama temporal

Entre cadascuna de les tasques existeix una relació de dependència. Algunes d'aquestes relacions fan que per començar una etapa sigui necessari haver completat l'etapa anterior. En altres casos, es pot començar un procés al mateix temps quan encara estem fent l'anterior. En la següent figura es mostra el diagrama temporal i, més endavant, s'explica detalladament:

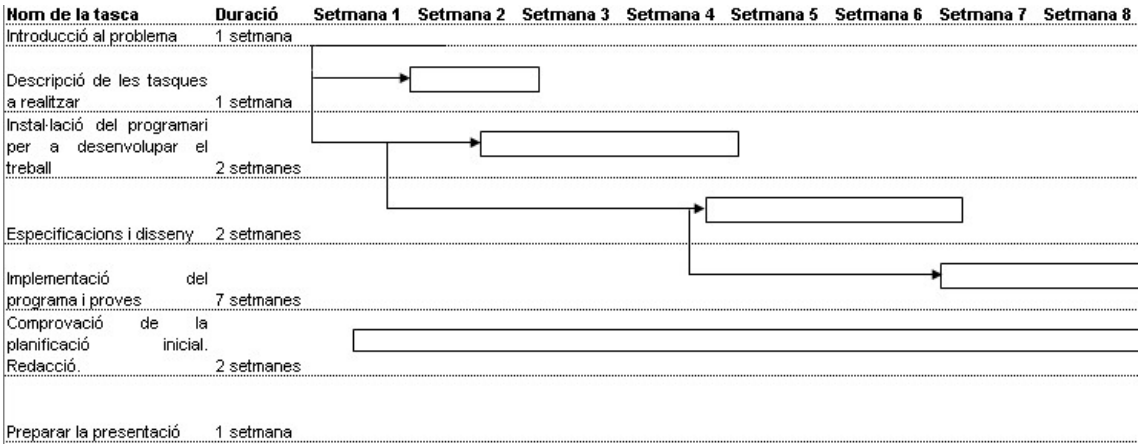

Fig. 1.1 Diagrama temporal

Un cop completada la part de la introducció, ja es poden començar dues tasques, la descripció de les feines i la instal·lació del programari. Aquests dos apartats no s'interfereixen, és a dir, que es poden desenvolupar al mateix temps.

Quan ja ens hem introduït a l'entorn de treball es comença a especificar el disseny, aleshores es coneixen moltes de les eines que s'utilitzaran. Però per la implementació de l'aplicació, és molt important haver acabat el disseny, i assegurar-se que estigui ben fet. És un requeriment que el disseny no contingui errors, car això comportaria problemes en el desenvolupament del programa

que potser no es podrien solucionar. Aquest fet ens podria obligar a fer un disseny nou del sistema, i el projecte no compliria el termini establert.

A mesura que es va creant l'aplicació i es fan proves, hi ha una recopilació d'informació per a la redacció. Totes les experiències que van apareixent, així com l'explicació de tot el que es fa, es redacten al moment de fer-les. Així doncs, el procés de redacció queda englobat temporalment en el temps d'implementació. Com s'ha explicat abans, també és una eina consultiva.

La tasca de preparar la presentació es fa l'última setmana, un cop finalitzada la redacció. Cal tenir tot el treball acabat, per tal que les transparències es puguin sintetitzar a partir del projecte d'una manera senzilla. És molt important fer una bona exposició, ja que és l'únic moment en què el Tribunal avaluarà l'autor en directe. Com s'ha dit anteriorment, s'ha de tenir un guió que contingui els punts a seguir durant la presentació, i també és útil fer un assaig abans del dia de l'exposició.

1.5. Glossari

Algorisme de *Dijkstra*: Algorisme d'encaminament que troba el camí més curt entre un origen i la resta d'elements del graf.

Graf: Estació de tren en aquest projecte. Està format per un conjunt de nodes o vèrtexs i els enllaços entre ells.

Circuit de via: Vèrtex, node. És la separació lògica de les vies. Quan un tren es situa en qualsevol punt d'un circuit de via arriba una indicació al centre de control tècnic (CCT). Es divideix en traços i agulles. L'inici del seu nom està compost per CVI, més l'identificador de circuit de via, més les sigles que corresponen a l'estació. És la unitat bàsica que es fa servir en l'aplicació, cada circuit de via representa un node.

Traç: Part d'un circuit de via, està separat de la resta de traços per una agulla. L'aplicació s'expressa en circuits de via per simplificar el procés, però els traços són els elements que necessita el CCT. Hi ha una base de dades per fer la conversió de circuits de via a traços.

Agulla: Element de la via que permet seleccionar entre dues sortides per una mateixa entrada.

Senyals: Semàfors de les vies.

Via: Aresta. Són les connexions entre els nodes. Poden tenir un pes, o cost del camí. En la present aplicació, tots els costos tenen valor 1.

Java: Llenguatge de programació orientat a objectes (OOB).

Sun Microsystems: Empresa propietària de l'arquitectura *Java*. És gratuït descarregar-se el programari, però només es pot fer de manera legal des de http:\\java.sun.com.

GUI: *Graphic user interface*. Interfície gràfica que fa que l'usuari/a pugui fer servir l'aplicació amb una plataforma gràfica.

JFC: *Java Foundation Classes*. Conjunt de classes de *Java* per a construir GUIs.

IDE: *Integrated Development Environment*. És el programari utilitzat per els/les programadors/es per poder crear, compilar i executar el codi de *Java*, en aquest cas. Es fa servir l'IDE *NetBeans* versió 3.6 de *Sun*.

Swing: Conjunt de components que es fan servir a les JFC. Aquest nom té relació amb el projecte inicial de JFC. Ara, el seu nom està present en els paquets *javax.swing.**, que contenen totes les classes.

AWT: *Abstract Window Toolkit*. Conjunt de components per a construir GUIs, de la mateixa manera que ho poden fer els components Swing. Hi ha algunes diferències entre els dos tipus de components, que s'expliquen en el capítol 2.

CAPÍTOL 2. METODOLOGIA

En aquest segon capítol hi ha tres punts claus: l'algorisme d'encaminament amb restriccions, el llenguatge de programació *Java* i el model per dissenyar sistemes de programació anomenat UML. Tots tres punts són els mètodes utilitzats per portar a terme aquest projecte.

2.1. Algorisme de *Dijkstra* amb restriccions

Per a entendre el funcionament de l'algorisme és necessari conèixer bé els seus elements, que es troben explicats en el glossari.

L'objectiu de l'algorisme és trobar el camí més curt (en termes de circuits de via), des d'un node origen donat a tots els altres nodes del graf, no només al node destí. El graf es denota per G=(V,E), està format per un conjunt de circuits de via (node o vèrtexs), i un conjunt de vies (enllaços, E). El node origen s'indica amb 'o' i el node destí s'indica amb 'd'.

Tot i que l'algorisme busca tots el camins des del node origen fins a la resta de nodes del graf, en aquesta aplicació no es farà servir d'aquesta manera. En el tràfic ferroviari es necessita conèixer el camí entre dos nodes donats, i cada itinerari es coneix un a un, per tant el programa trobarà el camins entre dos punts.

Al principi, el node origen considera que tots els nodes estan a una distància infinita, excepte ell mateix, que està a una distància 0 de l'origen (ell mateix). Existeixen dos conjunts anomenats 'S' i 'Q'. Al conjunt 'S' hi trobem els nodes pels quals ja s'ha trobat el camí més curt fins a l'origen, i al conjunt 'Q' hi trobem els nodes pels quals encara no s'ha trobat el camí.
Els dos conjunts requereixen una inicialització al principi de l'algorisme. S'ha d'inicialitzar 'S' de tal manera que no contingui cap node, i 'Q' l'hem d'inicialitzar per tal que contingui tots els nodes del Graf. Un cop inicialitzats, el que fa l'algorisme és extreure el node amb la mínima distància des de l'origen, aquest node s'indica amb 'u'. El primer cop que fem aquesta operació, extraurem el node origen, ja que com s'ha dit, aquest està a una distància 0 d'ell mateix, i la resta de nodes estan a una distància infinita. Aleshores aquest node amb mínima distància a l'origen passarà del conjunt 'Q' al conjunt 'S', esborrant-se de 'Q'.

En aquests moments apareix una funció anomenada *relax*, que cal explicar. Existeixen dos nodes: 'u' i 'n'. El node 'n' ja coneix un camí fins a l'origen, que creu que és el més curt. Si el camí per anar del node origen a 'n', és més llarg que, el camí que ha de fer un tren per anar de l'origen a 'u' i després a 'n', aleshores, s'estableix que el node precedent de 'n' és 'u', i es modifica el valor de la distància de 'n' des de l'origen.

Per a cada node del conjunt 'Q' que s'extregui es *relaxarà* respecte un node 'n'. El node 'n' s'extrau del conjunt 'Q'. Per a *relaxar* és indispensable que es compleixin dues condicions: que 'u' i 'n' siguin adjacents, o sigui, que estiguin connectats directament, i que no existeixi cap restricció entre ells. Tot això es repetirà fins que el conjunt 'Q' estigui buit.

Per tal d'entendre el funcionament de l'algorisme, s'ha expressat tal com es mostra en la figura de la pàgina següent:

```
algorisme{

        inicialitzar els nodes{
                //distància de l'origen a l'origen és 0.
                //distància de tots els nodes excepte o a l'origen és infinit.
        }
        inicialitzar els conjunts{
                //mètode per inicialitzar el conjunt S.
                //mètode per inicialitzar el conjunt Q.
        }
        mentre Q tingui elements{

                //mètode per trobar el node a mínima distància de l'origen.
                //mètode per esborrar el node trobat de Q.
                //mètode per afegir el node trobat a S.

                //anem agafant un node de Q (n).{
                        //mètode si són adjacents i si no tenen restricció.
                                //podem fer relax entre els nodes q i n.
                }
        }
}
```

Fig. 2.1 Explicació de l'algorisme de *Dijkstra* modificat.

2.1.1. Raons d'utilitzar *Dijkstra*

L'algorisme d'encaminament de *Dijkstra* ofereix tots els recursos que necessita un sistema de tràfic ferroviari. El més important, és el mecanisme de cerca del camí més curt entre un origen donat i la resta de nodes en un graf determinat. L'algorisme es modifica per afegir restriccions. Com es veurà més endavant, és possible afegir múltiples restriccions, tot i que en el projecte només n'hi ha una, s'explica com s'afegirien més restriccions.

2.1.2. Restriccions físiques i restriccions lògiques

La restricció existent en el sistema del projecte es basa en la impossibilitat que té un tren de seguir un itinerari en tots dos sentits. Per començar, tots els enllaços entre nodes són en ambdós sentits, per això s'anomenen també col·laterals. És a dir, un node té els mateixos enllaços d'entrada que de sortida, o el que és el mateix, té el mateix nombre de col·laterals.

Físicament un tren té un cap i una cua, o una part davantera i una part del darrere. Potser, totes dues parts poden anar al davant i al darrere, però mai en un mateix trajecte. Aleshores, un tren pot variar de direcció, com ho fa quan canvia de via mitjançant les agulles, però no pot canviar de sentit.

El programa ha de contenir aquesta restricció de manera lògica. El fitxer que té la informació del graf, té dades com el nombre de col·laterals i els identificadors d'aquests. En el següent exemple es mostra com l'aplicació sap que existeix una restricció:

- Hi ha tres nodes 'A', 'B' i 'C'. El tràfic va de 'A' a 'C' i prové de 'B'. En aquesta situació hi ha una restricció comuna a tots els enllaços del graf: el tràfic no pot anar de 'C' a 'A' ja que prové de 'A'. Es pot donar el cas que anar de 'A' a 'C' sigui impossible perquè el node 'B' no ho permeti. És a dir, que 'A' sap que quan el tràfic prové de 'B' no pot ni tornar a 'B' ni circular cap 'C'.

Per entendre millor aquests casos hi ha més exemples en el capítol 4. En aquests apartats hi ha unes figures que mostren amb claredat el concepte de restricció.

2.2. El llenguatge de programació *Java*

El llenguatge de programació *Java* és propietat de *Sun Microsystems*. És un llenguatge orientat a objectes.

2.2.1. Avantatges d'utilitzar *Java*

Entre els diferents avantatges d'aquest tipus de llenguatge, però fixant-nos només en els que té *Java*, citarem els que influeixen en el present programa:

- Subministra models similars als del món real.
- Facilita la capacitat per a tornar a utilitzar el codi.
- Abstracció.
- Modularitat.

La capacitat per a tornar a utilitzar el codi és de gran utilitat no només per aquesta aplicació, si no també per a futurs programes que comparteixin similituds. La construcció de classes i mètodes, així com de l'estructura (*framework*), permet que es pugui tornar a fer servir, modificant les parts que siguin necessàries, però no haver de començar de nou. Una gran part del temps de feina de implementació de l'aplicació, es destina a crear l'estructura del programa, en aquest cas, una estació de tren amb els seus elements.

Quan es fan servir objectes per primera vegada, s'ha de tenir clar que cada objecte existeix en el món real. Si no existeix, és perquè potser no és un objecte, potser és un mètode. Un objecte també es pot veure com una cosa que es pot comprendre intel·lectualment, com un procés. Una entitat de programari també pot ser un objecte.

Cada objecte té una identitat unívoca; un estat, conjunt d'atributs i un comportament, conjunt de mètodes. A partir d'això es poden donar vàries definicions, però els tres trets hi són presents.

No s'ha de confondre classe amb objecte, ja que un objecte és únic i pertany a una classe determinada. Un objecte existeix durant el temps d'execució i resideix en memòria. Una classe té un patró per a la definició d'un tipus d'objectes. Per exemple, la classe persona pot crear un objecte que es digui Guillem, de classe persona, que existirà el temps que s'executi l'aplicació.

Un punt important de la programació orientada a objectes recau en abstraure els mètodes i les dades comunes a un conjunt d'objectes i emmagatzemar-ho en una classe, és a dir, s'ha de saber quin comportament té cada element de l'estació, quines dades comparteix amb els altres objectes com ell, i crear una classe que recopili tota aquesta informació.

Totes les classes existents, no són independents. Existeixen unes relacions entre elles, que poden ser del tipus: associació; agregació i composició; herència i dinàmica. Més endavant s'explica les relacions entre les classes de l'aplicació.

2.2.2. *Swing* i AWT per a fer GUIs

Existeixen dos tipus de components per a fer GUIs. Cadascuna d'elles té unes propietats. En un principi, s'havia escollit els component AWT per a desenvolupar la interfície gràfica, ja que semblava més senzill per a desenvolupar el codi.

Es va buscar informació sobre tots dos paquets de components, i es va trobat informació sobre la idoneïtat d'utilitzar els components *Swing*. Aquests tenen un avantatge molt important: no incorporen codi natiu. Això significa que aquests components poden tenir més funcionalitats ja que no depenen d'un denominador comú, segons la plataforma. En la pròpia pàgina de *Sun*, es recomana l'ús de *Swing* per a fer GUIs.

A part d'aquest avantatge, també és possible modificar l'aspecte i el comportament del GUI en el cas d'utilitzar components *Swing*; a AWT l'aspecte del GUI depèn de la plataforma on es treballa.

2.2.3. UML, la notació

Són les sigles de *Unified Modelling Language*. És el successor dels mètodes d'anàlisi i disseny orientat a objecte de finals dels 80 i començaments dels 90.

És un llenguatge basat en diagrames que busca crear un model d'un sistema donat. Un model és una abstracció que aporta coneixement sobre el sistema. Aquest sistema acostuma a ser un problema, i el que es busca acostuma a ser una solució. UML és un llenguatge per a visualitzar (mitjançant diagrames) els elements d'un sistema de programari des d'una perspectiva orientada a objectes.

Els diferents tipus de diagrames UML són: de casos d'ús, de seqüència, d'objectes, de components, de desplegament, d'activitat, d'estats, de classes i de col·laboració.

Més endavant hi ha un diagrama de classe i un diagrama de seqüència sobre el problema de trobar el millor camí a seguir per un tren en una estació.

CAPÍTOL 3. CONTACTE AMB *JAVA*

En aquest capítol hi ha una breu descripció del programari utilitzat durant el projecte. Per a desenvolupar aplicacions *Java* són necessaris uns codis binaris, un *kit* per a desenvolupadors/es i documentació sobre les classes de *Sun*.

3.1. Programari necessari per al desenvolupament del projecte: J2SE de *Java* i l'SDK *NetBeans* 3.6

Per a poder programar amb la plataforma *Java*, cal tenir els binaris instal·lats a l'ordinador. Aquests binaris s'han de descarregar de la pàgina oficial de *Sun*. S'anomenen J2SE (*Java* 2 *Standard Edition*).

Juntament amb els binaris, cal un programari de desenvolupament per a programadors/es que s'anomena SDK (*Software Developer's Kit*). El paquet conjunt que inclou la J2SE i l'SDK s'anomena JDK i la versió descarregada és la 1.4.2 *Update 4*.

Finalment cal un IDE. La versió descarregada és la 3.6 de *NetBeans* (http:\\netbeans.org). Aquesta eina també es pot obtenir de la pàgina de *Sun*. És la interfície que et permet programar amb la plataforma *Java* de manera còmoda, amb moltes opcions disponibles.

3.2. APIs i tutors

API vol dir *Application Program Interface*. És un conjunt de rutines, protocols i eines per a construir aplicacions i programes.

Dins de la pàgina http:*Java*.sun.com hi trobem múltiples opcions per aprendre de tot sobre *Java*. En l'apartat de les especificacions de les API, es troba la documentació, on hi ha informació sobre totes les classes existents a J2SE, en la seva versió 1.4.2. Totes aquestes classes es poden utilitzar important la llibreria corresponent.

A més a més, en aquesta pàgina hi ha diversos tutors, que expliquen els diferents tipus d'aplicacions que es poden desenvolupar. Ho fa mitjançant exemples, explicacions, referències, etc.

3.2.1. Realització de proves

A mesura que es va pensant el disseny de l'aplicació, és necessari comprovar si allò que s'ha pensat, es pot portar a la pràctica. Per a simplificar el procés, es divideix en les parts més senzilles possibles i es proven parts de codi.

Per a provar aquestes unitats més senzilles, s'agafen exemples on s'utilitzi el codi necessari, i es creen petits programes per a veure si funciona. Un cop es veu el funcionament que té, els errors que pot generar i altres especificacions es poden fer dues coses. La primera és considerar aquell codi com a bo. L'altra opció és unir dos parts de codi, per provar una idea més complexa que la d'abans.

La majoria de proves fetes s'han fet seguint la mateixa metodologia. Es crea un programa amb un mètode principal. Aleshores es compila i s'executa com un programa autònom. L'únic objectiu no és veure com funciona el programa. A vegades és necessari extreure les excepcions que es llancen, per a poder-les capturar correctament en l'aplicació.

CAPÍTOL 4. ESPECIFICACIÓ I DISSENY DE L'APLICACIÓ

Al capítol quart s'explica el mètode que s'ha utilitzat per dissenyar l'aplicació. Hi ha dos diagrames fets a partir de la metodologia UML. Es descriu l'aplicació tal com va ser planificada, que va servir de base per a la posterior implementació del projecte.

4.1. L'aplicació

L'objectiu de l'aplicació es trobar el camí més curt entre dos circuits de via d'una estació. La persona que treballa al CCT necessita una eina, aplicació, que substitueixi el treball manual que està fent fins a l'actualitat. Aquesta persona ha de fer diferents passos per fer la feina:

- Conèixer l'estació on està treballant i aconseguir la base de dades que conté tota la informació referent a ella, com són els circuits de via, els traços, les agulles, etc.
- Trobar el camí més curt entre l'origen i el destí.
- Aquest camí es trobarà en relació a circuits de via, però el CCT gestiona també els traços i la posició que han de tenir les agulles i els senyals.

D'aquesta manera, qualsevol manera es pot equivocar. Per això té raó d'existir el programa d'aquest projecte. Dels tres passos anomenats a dalt, aquesta aplicació no els fa tots. Només és necessari conèixer els circuits de via i els seus enllaços per a trobar qualsevol camí. Per tant, la posició de les agulles i l'estat dels senyals no forma part d'aquest projecte.

El coneixement de l'estació també és indispensable; la informació es transmet de la base de dades al programa mitjançant un fitxer de text. Aquest fitxer de text s'ha comentat amb anterioritat, i es mostra un exemple real més endavant. Però aquí cal remarcar, que l'aplicació funciona per a qualsevol estació que segueixi els paràmetres marcats pel fitxer. Un CCT controla una estació habitualment, però com que existeixen casos d'emergència, pot ser que un centre de control hagi de controlar una altra estació. No es pot pensar en el tràfic entre estacions utilitzant aquesta aplicació, ja que això queda controlat en un altre nivell de seguretat del qual no tenim accés.

Ara ja es coneix què farà l'aplicació, i en quin ordre:

1. Tenir coneixement sobre l'estació de treball i construir el graf que la descrigui.

2. Escollir dos circuits de via, un d'origen i l'altre de destí. L'elecció és per part de l'usuari/a, no per part de l'aplicació.

3. El programa buscarà un camí entre tots dos nodes. El resultat serà, o bé el camí resultant, o bé un missatge d'errors que declara que no existeix cap camí possible amb els paràmetres establerts. Aquest últim cas es produirà quan s'incompleixi la restricció.

Existeixen dos modes de funcionament de l'aplicació que s'expliquen en els següents apartats. Cadascun dels modes té una execució diferent.

Per una banda, l'execució del mode comandes s'atura cada vegada que es porta a terme una operació. Una operació pot ser: trobar un camí, buscar un camí no possible, introduir malament els paràmetres, seleccionar un fitxer d'entrada incorrecte,...El resultat d'aquest mètode, si la recerca del camí té èxit, té forma de fitxer de text de la mateixa manera que en l'altre mode.

El canvi més important en l'aplicació que té interfície és justament la interfície. Aquest mode no té un temps d'execució definit. Depèn de l'usuari/a. Si la màquina on està corrent el programa funciona correctament, aquest només aturarà l'execució a petició de l'usuari/a. Més endavant hi ha un conjunt de figures que mostren pantalla a pantalla les opcions.

En aquest cas la persona encarregada del CCT pot anar buscant camins indefinidament només seleccionant l'origen i el destí. L'avantatge més destacat d'aquest sistema és la comoditat. Un desavantatge és la despesa de memòria que suposa. La interfície gràfica utilitza paquets *Swing* a diferència del sistema que funciona per línia de comandes. Cada paquet d'aquests afegeix memòria gastada.

Per aquest motiu s'ha dissenyat una interfície senzilla, sense afegir cap tipus d'imatge ni animació ni tampoc cap so. Les proves fetes amb els components *Swing* donen motius per pensar que l'aplicació no tindrà un cost de memòria suficient perquè l'usuari/a apreciï cap disminució de velocitat en l'execució.

El temps d'execució de l'algorisme i el temps de creació del graf són altres paràmetres a tenir en compte. En cap dels dos casos s'han apreciat problemes, tenint en compte que el graf està format per uns 90 nodes. Si el nombre de nodes augmentés considerablement valdria la pena fer proves per comprovar que l'aplicació no triga massa temps.

L'estació que s'ha fet servir d'exemple és la que té més nodes i més camins possibles de la ciutat de Barcelona. Tot i això, amb la velocitat dels ordinadors de l'actualitat, és improbable que l'aplicació d'aquest projecte tingui cap problema sigui quin sigui el tamany del graf.

4.1.1. Entrada i sortida de l'aplicació

L'aplicació necessita crear, a nivell lògic, l'estació. Per a fer-ho, ha de tenir unes dades específiques: nombre de nodes, enllaços de cada node,

identificador del nodes,... S'ha de recordar que un circuit de via representa un node.

Totes les dades necessàries es troben en una base de dades. Per fer menys feixuc el programa, es fa primer una conversió de les dades que hi ha a l'estació: senyals i traços, i les seves correspondències, a unes dades que s'emmagatzemaran a un fitxer de text. Aquestes últimes dades són: circuits de via i les seves connexions. L'aplicació utilitza un algorisme, i per tant, resulta més senzill calcular els camins d'un graf que només contingui les dades necessàries per a fer-ho.

El mateix passa amb el camí trobat. Ara és el CCT qui té la necessitat i l'aplicació qui proporciona una solució. S'utilitza un fitxer de text una altra vegada. Però ara amb un format diferent, ja que les dades són diferents.

Ara només necessitem saber per quins circuits de via passa el camí. Per fer la conversió d'aquestes dades, a una informació que pugui gestionar el centre de control, s'utilitza el mateix procediment que el fitxer d'entrada.

Aquestes conversions s'han fet amb l'empresa Telvent, ja que aquesta empresa treballa en aquest sector diàriament i és qui millor coneix els elements de l'estació.

Per a mostrar clarament el format dels fitxers. S'han creat dos fitxers molt senzills i imaginaris, que no corresponen a cap estació real.

Exemple de fitxer d'entrada en la següent pàgina:

```
/*Nombre de nodes*/ ←comentari
4 ←nombre de nodes al graf
CVICV7BR ←identificador del circuit de via
CVICV8BR
CVICV9BR
CVICV10BR
/*Node 0*/
3 ←nombre d'enllaços de sortida
3 ←nombre d'enllaços d'entrada
1 1 ←el primer és el node amb el qual connecta, el segon és el pes
2 1
F 1
/* RESTRICCIONS */
3 ←nombre de restriccions
1 2 1 2←el primer és el node del qual prové la restricció, el segon és el nombre de nodes als quals no podem anar, i el tercer és o són l'identificador/s dels node/s restringit/s.
2 2 1 2
F 1 F
/*Node 1*/
2
2
0 1
3 1
/* RESTRICCIONS */
2
0 1 0
3 1 3
/*Node 2*/
2
2
0 1
3 1
/* RESTRICCIONS */
2
0 1 0
3 1 3
/*Node 3*/
3
3
1 1
2 1
F 1
/* RESTRICCIONS */
3
1 2 1 2
2 2 1 2
F 1 F
```

Fig. 4.1 Fitxer d'entrada d'exemple

Aquest fitxer donaria el següent graf:

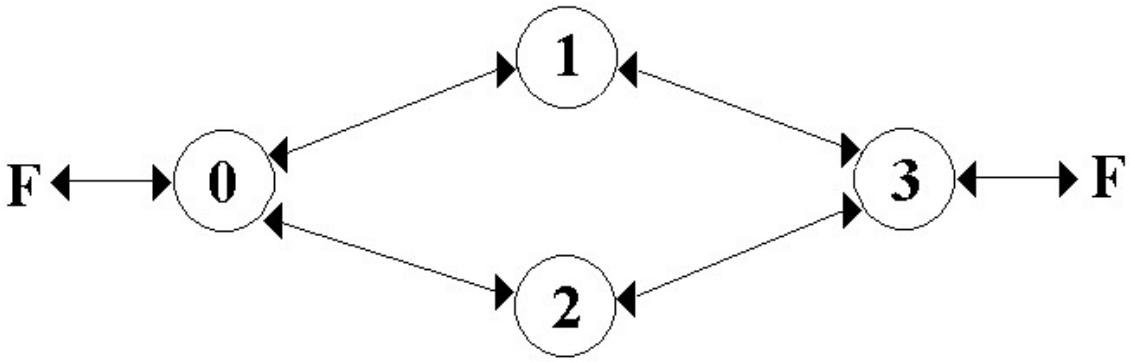

Fig. 4.2 Exemple de Graf

La lletra 'F' que hi ha a cada extrem del graf, significa Fora del Graf. És una representació a nivell físic, perquè es vegi que el graf és finit lògicament, però infinit físicament. És a dir, que l'aplicació no pot anar fins a 'F'. El tràfic entre estacions no es fa mitjançant la mateixa metodologia que el tràfic dins de l'estació. En el fitxer, el tràfic pot anar fins a 'F', suposant que 'F' és un node extrem.

Tots els enllaços són de dos sentits, per això tenim les restriccions. Els enllaços es mostren a la figura anterior, però les restriccions no. Per a veure-les s'exposa una altra figura.

Restriccions: exemple, si un tren provinent del node 1 va cap el node 0, té dues restriccions: no pot anar ni cap el node 1, una altra vegada, ni tampoc cap el node 2, ja que significaria retrocedir.

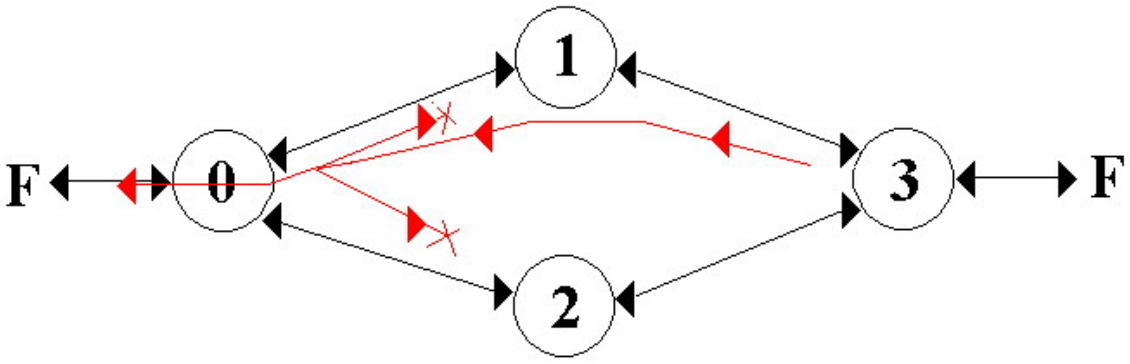

Fig. 4.3 Exemple de restriccions al Graf

Ara es mostra en una nova figura el camí que seguiria un tren que parteixi del node 0 per anar fins al node 3, el fitxer seria:

```
Node origen:
CVICV7BR
Node destí:
CVICV10BR
Camí a seguir:
CVICV7BR ←d'origen i el destí, tots dos inclosos
CVICV9BR
CVICV10BR
```

Fig. 4.4 Exemple de fitxer de sortida

En aquest exemple també es podria arribar per un altre camí del mateix cost. L'altre camí seria **Node 0 → Node 1 → Node 2**. En aquest cas no té importància escollir un camí o l'altre, ja que l'única condició és trobar un camí.

4.1.2. Interfície gràfica i línia de comandes

La part més important de l'aplicació és l'algorisme. Si el càlcul del camí és incorrecte, o no es pot calcular, l'aplicació no serveix. Tot i això, la forma de fer servir l'aplicació, és a dir, la interacció amb l'usuari/a ha de ser eficient, correcta i senzilla.

La condició que sigui correcta és necessària en les dues modalitats que s'han pensat.

Per a fer-ho senzill, es pot implementar una interfície gràfica. Compleix la condició de ser correcta. Dins de l'apartat de l'eficiència, s'ha procurat fer una interfície senzilla. En un CCT, no cal tenir una interfície que ocupi molta memòria en favor de l'estètica. Es pot tenir una estètica acurada, sense carregar la memòria.

Per altra banda, si volem guanyar eficiència, es pot fer una variant del codi anterior. És a dir, executar el programa des de la línia de comandes. Existeix la possibilitat que una màquina del CCT no suporti la interfície gràfica. Si bé es cert, que els binaris J2SE de *Java* són necessaris, el programa en execució gastarà menys memòria que la versió anterior, amb interfície gràfica.

En totes dues versions, la sortida on hi ha la informació del camí, serà un fitxer de text. Això fa que no hi hagi cap problema per la seva lectura en cap de les dues modalitats.

4.1.3. Captura d'excepcions (errors)

A *Java* existeix una màquina virtual que s'encarrega de gestionar la memòria. Un/a programador/a no ha de preocupar-se de l'accés a memòria quan està fent una aplicació amb *Java*. De la mateixa manera, *Java* controla els possibles errors en el codi, perquè quan hi hagi una excepció, això no faci finalitzar anormalment el programa.

Les excepcions que van ocorrent són llançades per la màquina virtual a l'espera que el programa les capturi. Hi ha dues maneres de fer-ho, tot i que la captura de les excepcions és obligatòria. Una manera, és capturar les excepcions i prou. L'altre manera, és capturar les excepcions, saber per què passen i preparar un codi que s'executi quan passi això.

Un exemple molt clar i senzill per entendre el tema de les excepcions és la lectura o escriptura d'un fitxer. Aquest cas succeeix en l'aplicació. Quan s'obre un fitxer poden passar vàries coses: la lectura és correcta, el fitxer no es pot obrir, no es troba el fitxer,... Si es vol que quan es capturi un d'aquests problemes, com per exemple quan no es troba el fitxer, no s'acabi el programa, es pot modificar el codi de tal manera que l'usuari/a sàpiga què ha passat i pugui introduir un fitxer nou.

En la implementació de l'aplicació, per a capturar les excepcions correctes, es van haver de fer proves. Aquestes proves consistien, moltes vegades, en provocar els errors, i així veure les excepcions que ocorrien. Aleshores, per tal de mostrar aquests errors, i que es fa amb ells, hi ha un conjunt de fitxers erronis, on es comprova que tot funcioni correctament.

La utilitat de tot això, és per comprovar que els fitxers que es fan servir d'entrada siguin correctes. També és evident, que si l'aplicació no pot llegir el fitxer, és perquè aquest no és correcte. Però pot ser que una estació tingui 100 nodes, com en el cas del fitxer que s'ha fet servir per a les proves. El volum de línies que ocupa és elevat; en cas d'error, hauríem de buscar en el fitxer quin és el problema. És molt útil que el programa indiqui a quina zona del fitxer ha localitzat el problema. Si bé és cert, que la correcció del mateix s'haurà de fer manualment.

Alguns errors possibles amb la lectura de fitxers són els següents:

- Error dins d'algun node del fitxer, provocat per un mal tecleig o problema semblant. El programa diu a quin node pot estar l'error.

- Error de format del fitxer, si el fitxer no és de text, el programa dóna un avís. En aquest cas hi ha una coincidència a l'hora de capturar els errors. Es poden tenir: un fitxer d'un format que no sigui de text i un fitxer de text que contingui la informació d'un graf. Si aquest segon fitxer té un error al node 0, pot donar el mateix error que al llegir el fitxer que no és de text. Per tant, s'haurà de tenir en compte aquesta condició.

- També existeixen errors de validesa en el graf. És a dir, el fitxer té el format correcte, però part de la informació que conté no correspon a un graf vàlid. Si un node no està connectat a cap altre, ningú coneix com arribar a ell. Això només donarà problemes si el node és l'origen o el destí d'un camí. Exemple: si el node 5 està enllaçat amb els nodes 31 i 36, es busca que els nodes 31 i 36 tinguin com a enllaç el node 5. De la mateixa manera es pot comprovar amb les restriccions.

4.2. UML

El diagrama de classe mostra les classes que formen part del sistema i la relació entre elles. En canvi, el diagrama de seqüència mostra la interacció entre els objectes.

4.2.1. Diagrama de classes

El diagrama de classes representa el disseny de l'aplicació. De manera gràfica, estructura les classes segons les seves relacions.

La forma de mostrar les classes és jeràrquica. La que està més amunt és la classe principal, que s'anomena **DijkstraModificat**. Tot i que aquesta classe és diferent en l'aplicació que s'executa amb interfície gràfica, que la que s'executa amb la línia de comandes, l'estructura i el nom són coincidents.

A cada classe li correspon un quadre, on hi apareix el nom de la classe a la part superior; a la part inferior hi ha els mètodes i els atributs que corresponguin.

La unió de les classes es fa mitjançant línies contínues, ja que totes elles tenen una relació d'ús. Les línies discontínues es farien servir si alguna classe implementés alguna interfície.

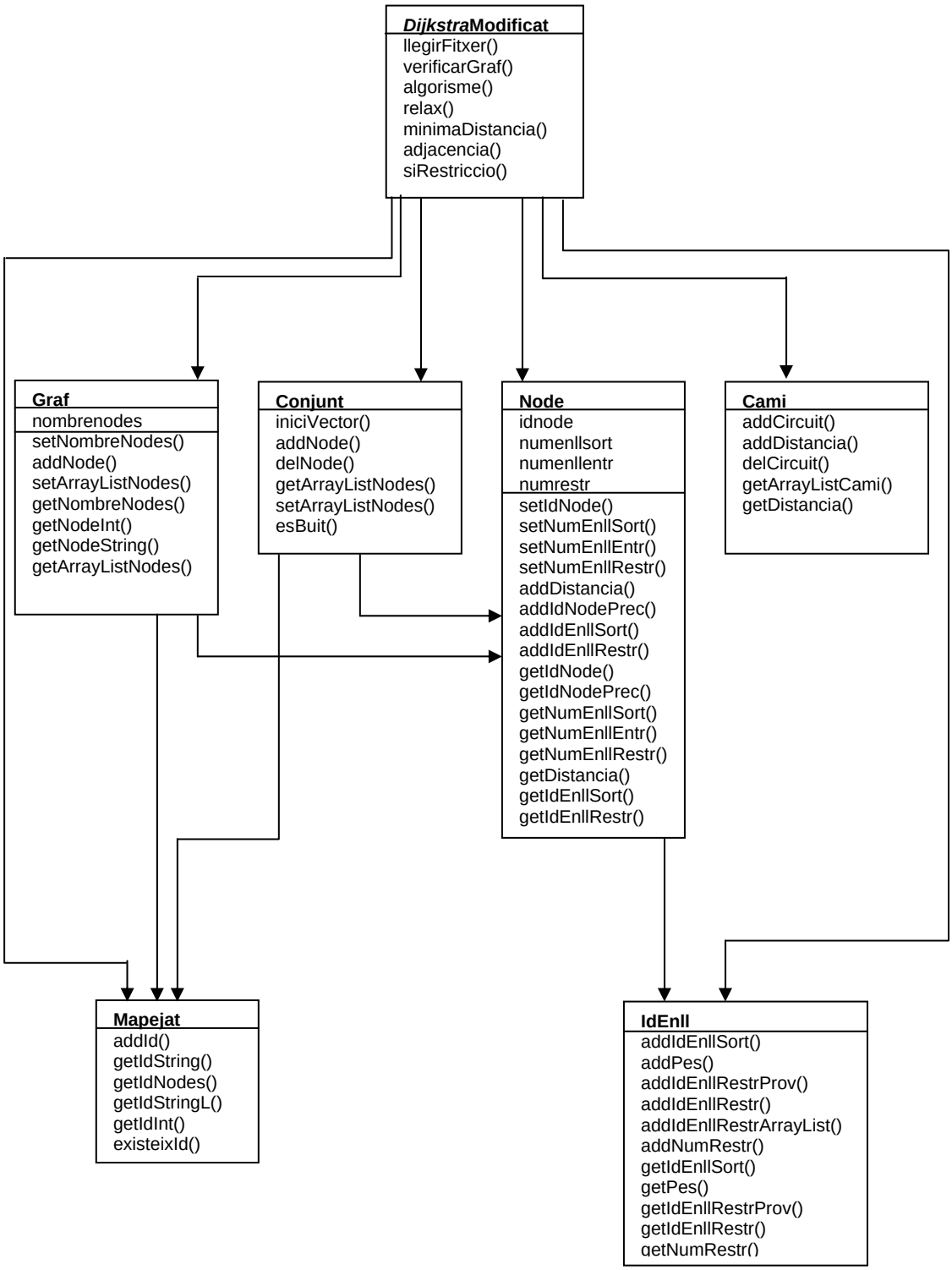

Fig. 4.5 Diagrama de classes

4.2.2. Llistat de les classes

En aquest apartat es llisten les classes utilitzades en l'aplicació. Per tal de veure una descripció detallada cal llegir l'annex.

- **DijkstraModificat**: Aquesta classe conté el mètode *main* o mètode principal. Té els mètodes de llegir el fitxer, aplicar l'algorisme i crear la interfície gràfica. Fa servir totes les altres classes.

- **Graf**: Aquesta classe crea l'estació de manera lògica. Fa servir la classe **Mapejat**.

- **Conjunt**: Aquesta classe es fa servir per l'algorisme de *Dijkstra* modificat. Fa servir la classe **Mapejat**.

- **Node**: Aquesta classe conté la tota la informació sobre un circuit de via. Fa servir la classe **IdEnll**.

- **IdEnll**: Aquesta classe es fa servir per estructurar els enllaços dels nodes, tant pels de sortida, com pels enllaços restringits.

- **Cami**: Aquesta classe emmagatzema el camí trobat.

- **Mapejat**: Aquesta classe conté una relació entre els identificadors dels nodes i la posició que ocupen en l'ArrayList.

4.2.3. Diagrama de seqüència

A la següent pàgina es mostren les diferents accions que pot fer l'usuari/a. En cada quadre hi apareix el nom. Cada acció comporta uns processos diferents.

El diagrama està expressat de manera que l'execució segueixi el procés correcte. Si es porta a terme una acció que no toca, simplement apareix un missatge d'error a la pantalla, i l'aplicació continua en el mateix punt. Aquests missatges d'error no apareixen al diagrama, per no omplir innecessàriament el gràfic.

Els passos que si que apareixen són els de retrocedir. És a dir, que independentment del punt on es trobi el programa, si es vol tornar al principi del procés, o a un pas intermedi, es pot fer. Les línies que expressen un moviment cap endavant són contínues, i les línies que ho fan cap enrere són discontínues.

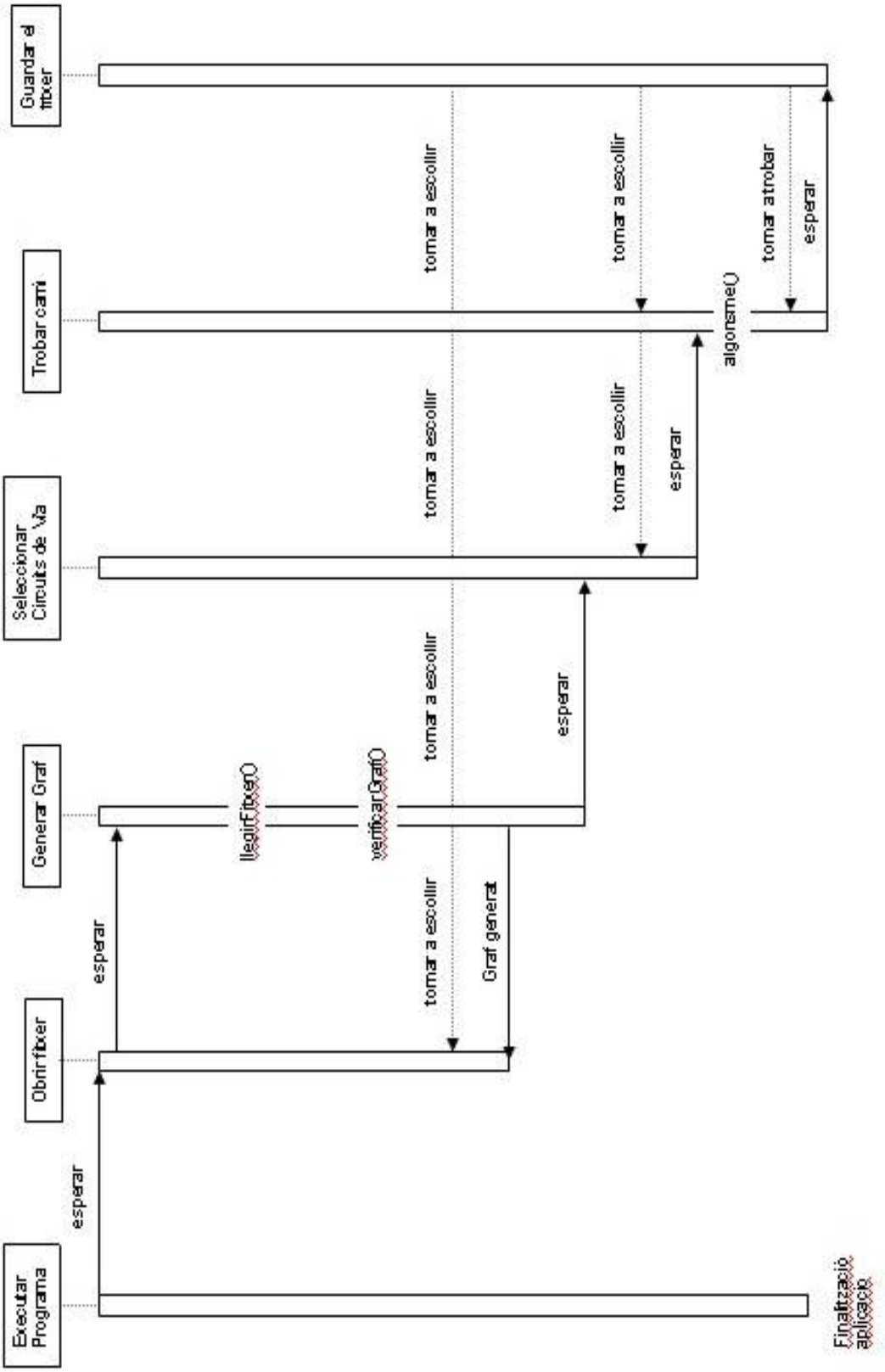

Fig. 4.6 Diagrama de seqüència

CAPÍTOL 5. DESENVOLUPAMENT I PROVES

En aquest treball hi ha dues modalitats operatives. En el cinquè capítol es descriuen els dos modes de funcionament. Es mostra pas a pas, mitjançant figures, el procés que segueixen per tal d'executar l'algorisme. La descripció dels dos models es fa juntament amb els comentaris que acompanyen les figures.

S'explica també, com es capturen les excepcions, quins errors són controlats pel programa. Hi ha exemples que descriuen de quina manera s'ha comprovat el format i la validesa dels fitxers utilitzats.

5.1. Interfície gràfica

El mode de funcionament de l'aplicació en interfície gràfica té sempre una finestra en execució. Quan l'usuari/a executa l'aplicació en aquest mode, li apareix una finestra com la que mostra la figura:

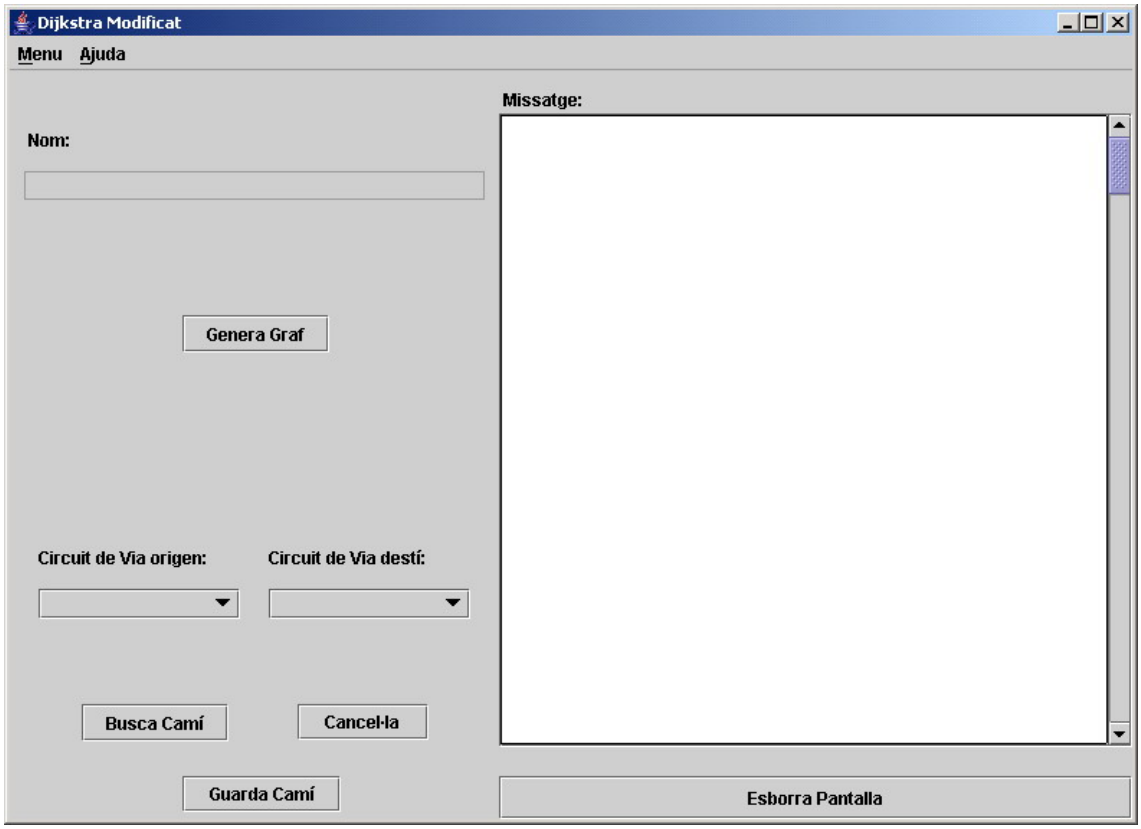

Fig. 5.1 Finestra principal

Aquesta finestra es pot tancar de vàries maneres, depenent del sistema operatiu on es treballi, i com estiguin configurades les opcions de finestra. Si es tanca la finestra, s'acabarà l'execució de l'aplicació.

Per exemple, si es polsa la creu de dalt a la dreta, s'enviarà un esdeveniment al programa per avisar-lo que l'usuari/a vol tancar-lo. Des del menú també es pot tancar, accedint a 'Menú'→'Tancar'.

La lletra que surt subratllada, és la que té drecera de teclat. Sigui quina sigui la manera de tancar l'aplicació, ens apareixerà la següent finestra:

Fig. 5.2 Finestra per sortir de l'aplicació

Abans de començar a fer servir l'algorisme, es pot continuar mirant les opcions de finestra. Si s'accedeix a 'Ajuda'-->'Sobre'. Aleshores apareix una finestra, que dóna informació breu sobre el GUI que estem fent servir:

Fig. 5.3 Finestra sobre l'aplicació

Aquesta finestra apareix sobreposada a la finestra principal. Si bé es pot seguir utilitzant la finestra principal, l'altre quedarà en execució. La manera que desaparegui la finestra emergent és prement el botó. Això succeeix amb totes les finestres emergents que s'expliquin a partir d'ara.
Dins del submenú 'Ajuda', es troba també 'Instruccions'. És el que mostra la següent figura:

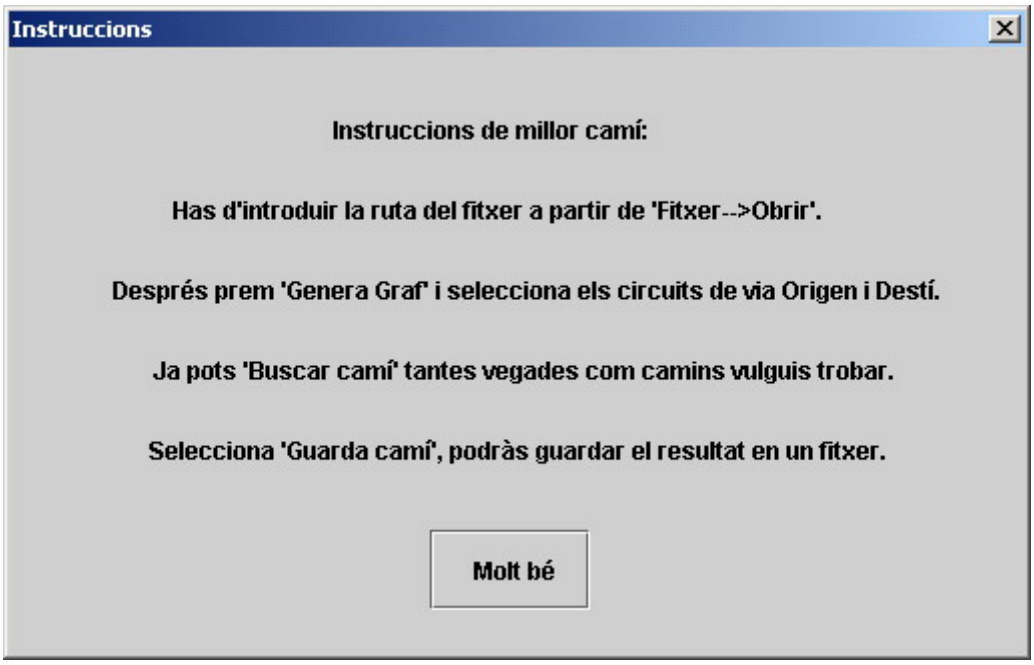

Fig. 5.4 Finestra sobre l'aplicació

El primer que cal tenir per a fer servir l'aplicació és el fitxer que conté la informació de l'estació. Per obrir-lo s'ha d'accedir a 'Menú'→'Obrir'.

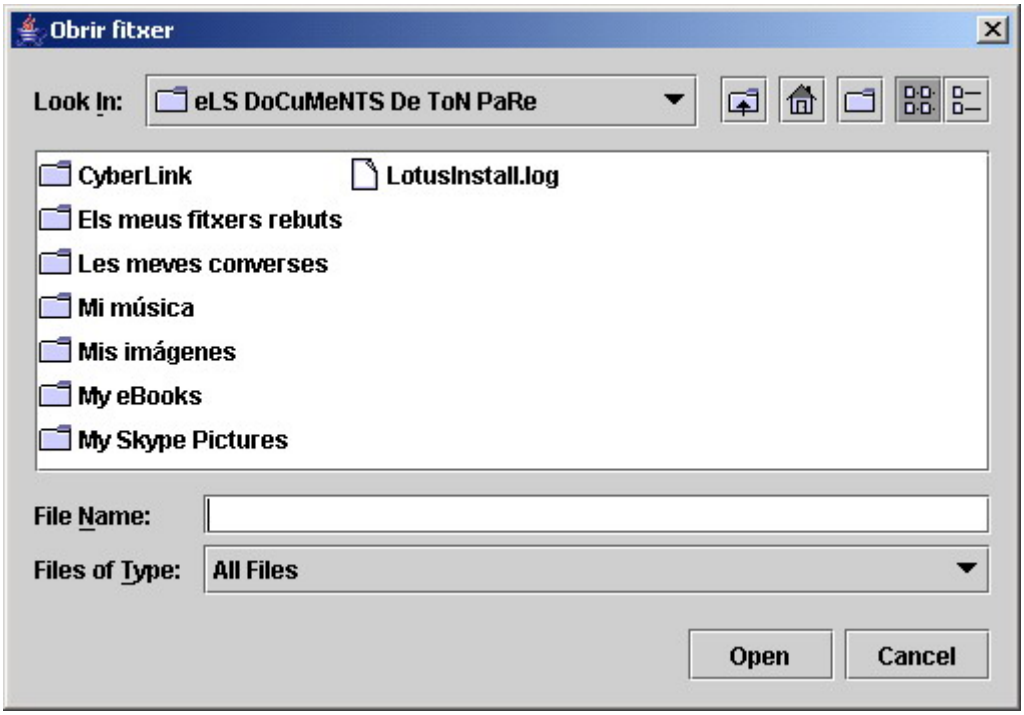

Fig. 5.5 Finestra per obrir un fitxer

El fitxer d'entrada pot estar a qualsevol ruta. No té cap importància.

A partir d'aquest punt s'han de seguir els passos següents (no s'insereix una figura a cada pas perquè no fa falta i perquè ocupa molt d'espai).

1. Prémer el botó de 'Genera Graf'

2. Seleccionar els circuits de via.

3. Prémer el botó 'Trobar Camí'.

A cada pas apareix un missatge al quadre de text, que es veu en la següent figura. Per a seguir trobant camins dins d'un mateix Graf, no cal tornar a generar-lo, tot i que si ho fem no hi ha cap problema:

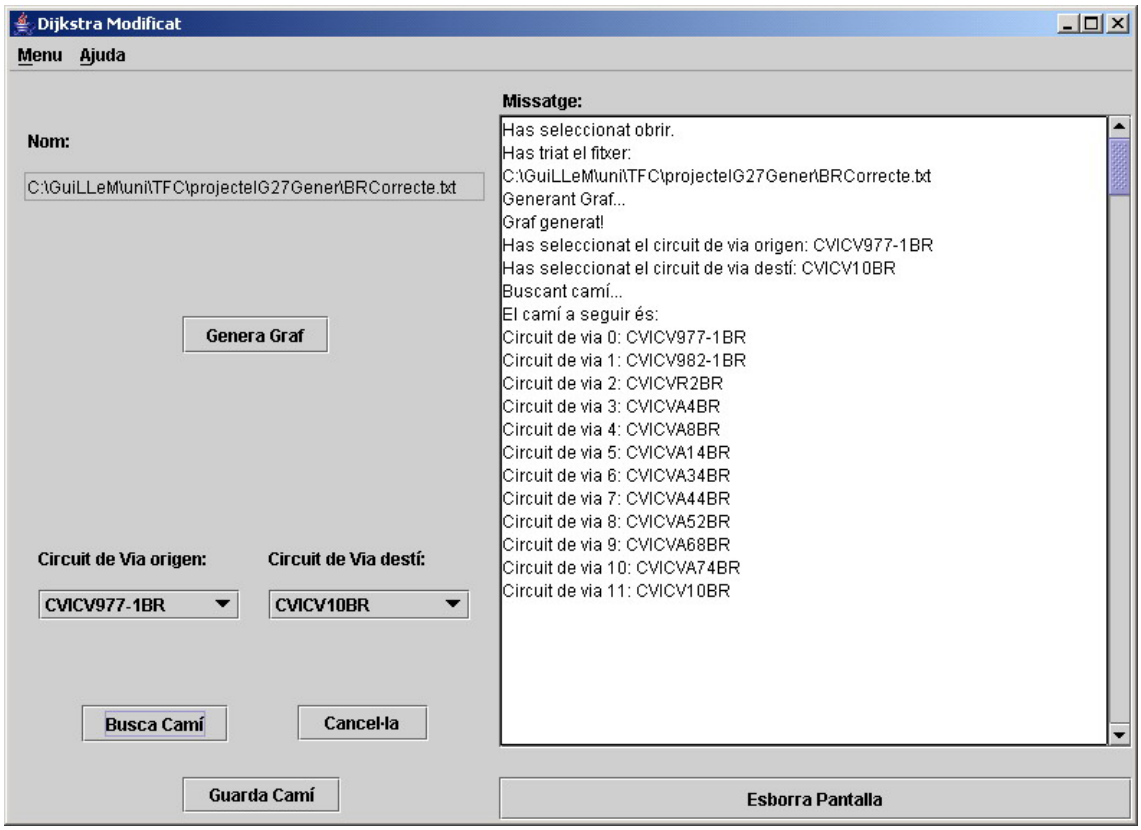

Fig. 5.6 Finestra després d'haver trobat un camí

Pot ser que no interessi guardar el resultat en un fitxer, ja que el resultat apareix per pantalla. Però el més habitual serà guardar-ho en un fitxer de text. Per fer-ho només cal accedir al botó de la pantalla principal, 'Guarda Camí'.

Quan es fa, apareix una pantalla pràcticament igual que per obrir un fitxer. L'extensió '.txt' és la necessària i, a més a més, s'ha comprovat el seu funcionament.

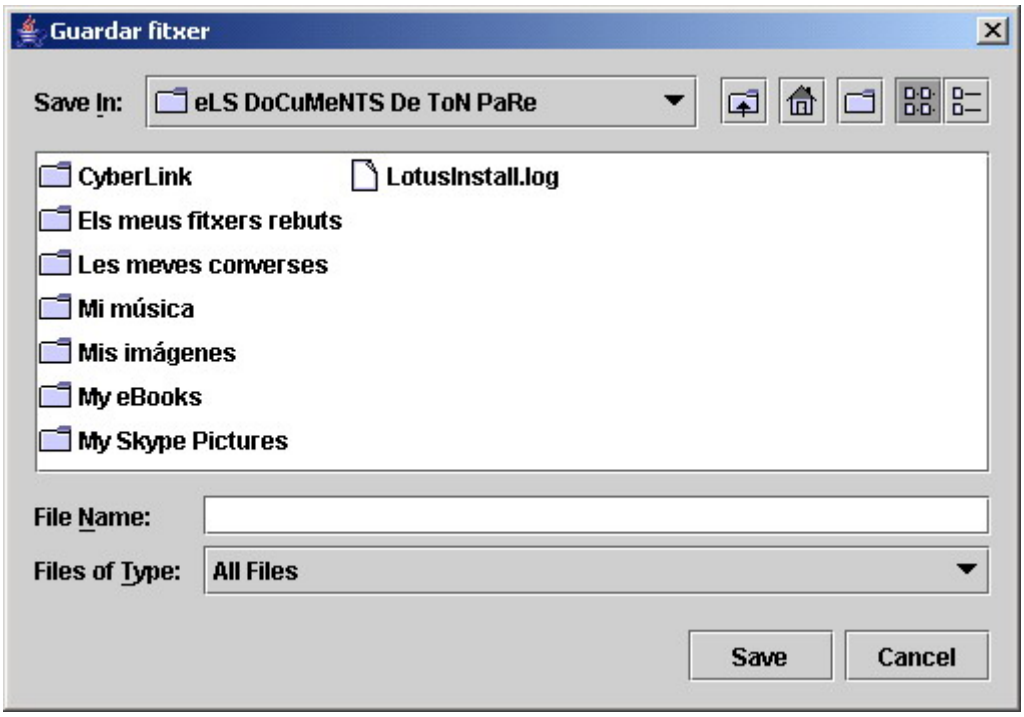

Fig. 5.7 Finestra per a guardar el fitxer

Un cop s'ha arribat aquí, es pot tornar a repetir les operacions que es vulguin. De fet es poden fer operacions correctes i operacions incorrectes. Aquestes últimes, estan controlades. Quan es fa una operació incorrecte apareix un text al quadre de text. En aquest programa es tenen en compte els següents problemes:

- A la pantalla inicial si s'intenta generar el graf, apareix el missatge:

  "*Error llegint el fitxer.*
  Introdueix un fitxer."

- Si a qualsevol moment s'intenta trobar un camí sense haver generat el graf, apareix el missatge:

  "*Buscant camí...*
  El Graf no està generat. Genera'l abans de trobar el camí."

- Si a qualsevol moment s'intenta guardar un camí sense haver-lo trobat, apareix el missatge:

  "*Primer has de trobar algun camí*."

- Si ja has generat el graf, però no s'ha seleccionat cap o només algun circuit de via, i s'intenta trobar un camí, veus el següent:

"*Buscant camí...*
Has d'introduir algun circuit de via per l'origen i pel destí"

5.2. Línia de comandes

Aquesta variació de l'aplicació és idèntica a l'anterior en el fons, però diferent en la forma. A vegades es fan servir màquines que tenen un sistema operatiu sense interfície gràfica, sobretot en sales de control. El que més importa en centres de control és l'eficiència i la robustesa de les aplicacions, deixant de banda l'aspecte que tinguin.

Aquesta aplicació en la versió de línia de comandes compleix els dos requisits. Per a executar aquest mode de programa es necessita el mateix programari que s'ha fet servir en la versió anterior. Per a poder executar el programa, cal anar al directori on hi hagi els fitxers guardats, i escriure la següent línia en l'intèrpret de comandes:

java DijkstraModificat fitxerorigen circuitorigen circuitdestí fitxercami

com es mostra també a la figura:

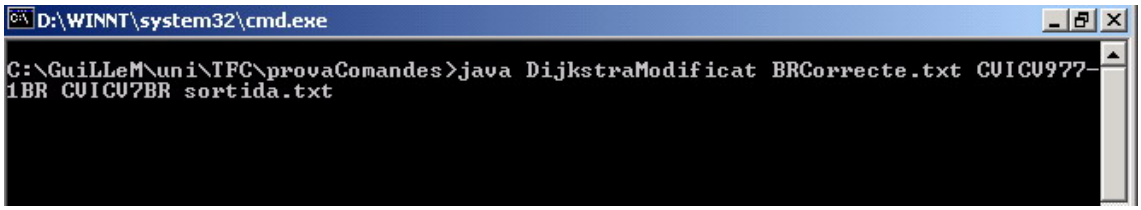

Fig. 5.8 Línia de comandes per executar el programa

L'aplicació *Java* funciona des de qualsevol directori on s'executi, no cal estar al mateix directori on està instal·lat el paquet J2SE. **DijkstraModificat** és el nom de l'aplicació. Cal respectar les majúscules i les minúscules. En aquest cas, si que és necessari estar al mateix directori on es trobi el programa i tots els seus arxius (extensions *java* i *class*). Els fitxers amb extensió *class* es creen quan es compilen els arxius *java*. S'inclouen els fitxers ja compilats.

Resta explicar el funcionament dels paràmetres. N'hi ha 4. És obligatori indicar els quatre paràmetres; si no es fa així apareix el següent missatge a l'intèrpret:

"*Recorda que has d'inicialitzar el programa així:*
'DijkstraModificat fitxerorigen circuitorigen circuitdesti fitxercami'"

El primer paràmetre és el fitxer d'entrada de dades. S'ha d'introduir la ruta completa d'on es troba el fitxer. En l'exemple, només apareix el nom, ja que el fitxer es troba al mateix directori. També s'han de respectar aquí les majúscules, si bé pot ser que en alguns sistemes operatius no faci falta. Els errors que pot originar la lectura del fitxer, es tracten més endavant.

El segon i el tercer paràmetre es refereixen als circuits de via. S'ha de posar el nom complet de cada circuit. El programa verifica que existeixin els dos. Si un dels dos no existeix, apareix un missatge per pantalla:

"*El circuit destí que has introduït no existeix en el Graf que has seleccionat.*"

"*El circuit destí que has introduït no existeix en el Graf que has seleccionat.*"

L'últim paràmetre també és un fitxer. En aquest cas, és un fitxer que pot no existir. Si no existeix, el programa el crea. Es pot especificar la ruta, però per defecte serà el directori de treball. Si el fitxer ja existeix, se sobreescriurà la informació sense preguntar res a l'usuari/a. Si es vol conservar el fitxer, només cal modificar el nom del fitxer.

Aquí no apareixen problemes amb l'ordre d'execució com en el cas de l'aplicació amb interfície gràfica. Com s'ha explicat en la descripció de l'aplicació, el mode de comandes funciona d'una sola vegada en tots els casos. Sigui quin sigui l'ordre que s'enviï a l'intèrpret, sempre es retorna altre vegada a l'intèrpret. El propi programa d'aquest projecte s'encarrega que no hi hagi conflicte com: intentar generar el graf sense haver llegit el fitxer o intentar guardar un camí sense haver-ne trobat cap, ...

El resultat de la búsqueda d'un camí es guarda en un fitxer de sortida, com s'ha vist. Però també s'imprimeix per pantalla, i queda de la següent manera:

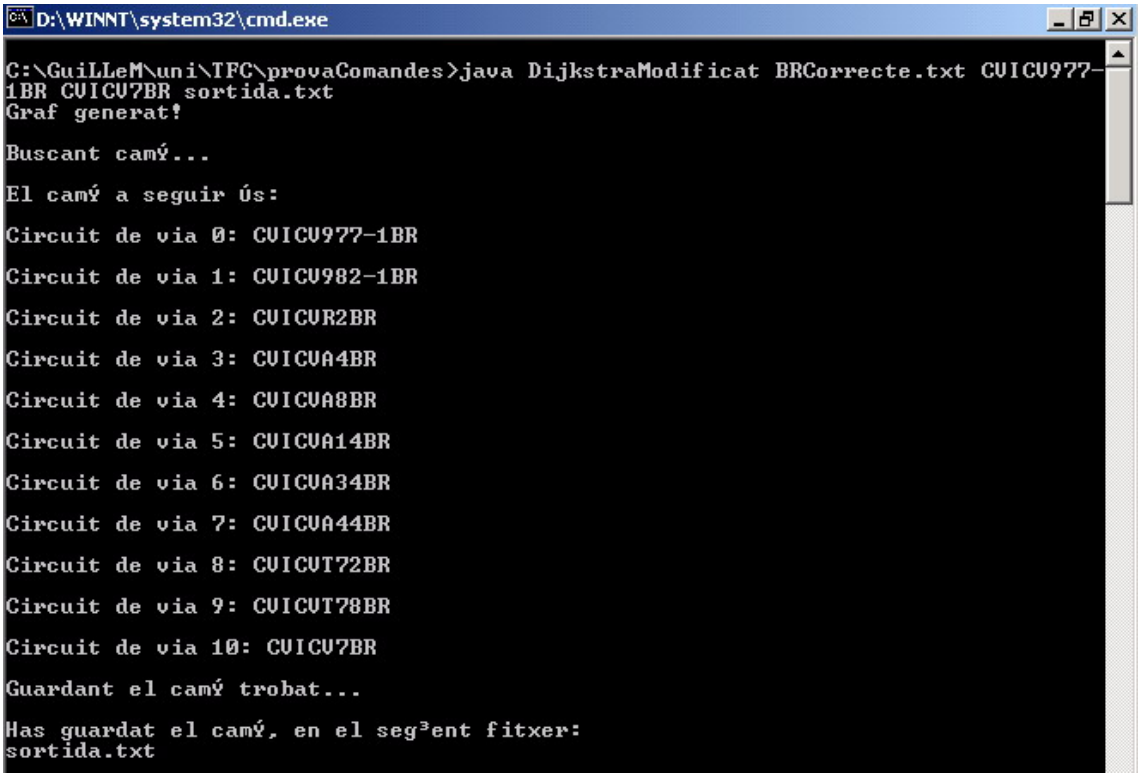

Fig. 5.9 Línia de comandes després de trobar un camí

Es pot observar alguns problemes de gramàtica en el textos que apareixen a les figures. Això es deu a l'ús d'accents. Aquests accents no afecten al programa, ja que les variables tenen noms sense accents. A l'hora d'escriure comentaris, he decidit escriure amb accents, ja que és com s'ha de fer. A més a més, alguns sistemes operatius segons com estiguin configurats, ho accepten sense problemes. Afegeixo, que si fossin algun obstacle per al funcionament de l'aplicació, no s'utilitzarien.

5.3. Captura d'excepcions

Un avantatge d'haver programat amb *Java*, és que els errors que podrien provocar una parada del programa, es capturen. Un cop capturats, es pot aplicar un codi en temps d'execució i no parar l'aplicació.

Com s'ha vist en l'apartat anterior, el programa pot controlar alguns problemes que puguin sorgir. Però cap dels exemples que s'ha mostrat anteriorment, són errors que provocarien una aturada del programa.

En els casos que explico ara, s'aprofita una de les qualitats de *Java*. De fet, el propi llenguatge obliga a capturar els errors, que en un moment donat podrien aturar el programa. Així s'evita que hi hagi una interrupció brusca. Opcionalment es pot afegir un codi per executar.

Hi ha dos punts on es poden produir excepcions en aquesta aplicació: en la lectura/escriptura dels fitxers i en la construcció del camí.

En el cas dels fitxers, les excepcions poden ser degudes a: que no es pugui obrir el fitxer de lectura (perquè no és un fitxer de text), que no pugui escriure en el fitxer de sortida, que no pugui crear el fitxer de sortida o que en el procés de lectura, s'intenti llegir un format de fitxer incorrecte.

En el cas de la construcció del camí, el problema ve d'intentar accedir a una dada que no existeix. És a dir, com que s'està intentant seguir un camí inexistent, una de les variables referents a un dels nodes, no té el valor adequat perquè no té valor. Aquest és el fet que produeix l'excepció.

5.3.1. Errors de format en els fitxers

A mesura que avançava el desenvolupament del programa va aparèixer una nova necessitat. El fitxer conté les dades per a la formació del graf. L'estació que s'utilitza per a fer proves té al voltant de 90 nodes, per tant, el volum de línies que conté és considerable.

Durant les proves fetes amb l'aplicació, es va detectar un error en una línia del fitxer. Era un error de conversió de dades a partir de la base de dades que conté la informació del graf. Va costar molt de temps detectar-lo, i per a no haver de tornar a repetir el temps de recerca d'errors en una altra ocasió, es va buscar un mètode que ho fes.

Existeix un mètode per llegir el fitxer que es diu **llegirFitxer**. Aquest mètode llegeix el fitxer, que està ordenat node per node. Això vol dir que ha d'acabar de llegir un node per començar a llegir el següent. Aleshores, s'introdueix una variable global, que s'incrementi cada vegada que s'acaba de llegir un node. Així, el programa recordarà a quin node s'ha aturat el procés de lectura quan capturi l'excepció. Per altra banda, s'ha de controlar l'excepció al cridar el mètode.

L'excepció que es captura és del tipus: *NumberFormatException*. Significa que una de les variables que es llegeix no és del tipus correcte. Això pot passar per diversos motius, com que falti un número, o una línia sencera. S'ha de recordar, que la lectura del fitxer ha de ser precisa. El format és únic.

Per a poder provar millor tot el codi nou afegit, es van crear uns fitxers nous amb errors. Com que els errors es creaven segons la necessitat, també calia modificar el nom dels fitxers d'aquesta manera: BRerrorEnllaçNode87.txt, BRerrorNode0.txt, BRerrorNode58.txt i BRerrorRestrNode71.txt.

Els resultats d'aquestes proves donen el mateix resultat i la mateixa sortida per pantalla. L'aplicació no s'atura, però és necessari introduir un fitxer nou per seguir amb el funcionament habitual del programa. Naturalment, en la versió de la línia de comandes, el que s'ha de fer és introduir de nou la comanda utilitzant un altre fitxer.

L'origen de l'error pot ser que tot i que el fitxer s'hagi pogut obrir (perquè és de text, per exemple) no contingui la informació de cap graf. Aleshores, al començar a llegir, es llançarà l'excepció. El comptador de nodes llegits estarà a 0.

De la mateixa manera, si es llegeix un fitxer de text que si que conté informació d'una estació i té un error node 0, el comptador de nodes llegits també estarà a 0.

Aleshores, en tots dos casos, el missatge per pantalla serà aquest:

"*Error llegint el fitxer. Per un error de format o un error al node: 0*"

Si l'error es produeix més enllà del node 0, es dedueix que el fitxer llegit si que conté dades sobre un graf. Aleshores per pantalla apareix el següent text, en totes dues versions (línia de comandes i interfície gràfica):

"Error llegint el fitxer. Per un error de format al node: 35"

Les probabilitats que existeixi un fitxer de text que tingui la informació correcte d'un graf fins al node 0 per casualitat són molt baixes. Per això se suposa que un cop llegit el node 0, el fitxer conté dades sobre una estació..

5.3.2. Errors de validesa en els fitxers

Els problemes que van sorgir arran de les proves van continuar. Hi havia un camí, que l'algorisme no era capaç de trobar. La resta de camins del graf, els trobava tots, i a més, de manera correcte. Com que s'havia revisat ja el format del fitxer, gràcies a les millores explicades a l'apartat anterior, el problema semblava venir de l'algorisme.

Es van observar altres nodes que estaven en la mateixa situació dins del graf, que el node afectat. És un node extrem, però la resta de nodes extrems, si que complien l'algorisme. Aleshores, es va mirar el fitxer, i es va comparar els enllaços del node, i dels nodes als quals aquest estava connectat, havien

d'estar connectats a ell. S'ha dit al principi, que tots els enllaços funcionen en els dos sentits.

Per entendre-ho, si el node afectat era el node X, i estava connectat amb els nodes Y i K, ni el node Y ni el node K sabien com arribar a X. Per tant, era impossible traçar un camí fins a X, perquè ningú sabia arribar-hi.

Això provoca un error de validesa del graf, i no de format. Semblava més difícil de detectar, un cop va aparèixer. Però tal com s'ha explicat, es veu ràpidament la solució. S'ha de crear un mètode, que verifiqui que els enllaços del node X, siguin correspostos pels seus nodes enllaçats.

A continuació es mostra un exemple d'aquest error de validesa. El fitxer que s'ha fet servir per a aquest cas es diu BRerrorEnllaçNode87.txt:

"Graf no generat. Hi ha un error de validesa.
Comprova el node 87 i els seus enllaços."

Ampliant aquesta verificació, es poden validar les restriccions utilitzant la mateixa metodologia. D'aquesta manera apareix un mètode anomenat: **verificarGraf**.

Per a fer proves amb aquest mètode nou, també es van crear fitxers modificats expressament, sabent on estava l'error. El fitxer utilitzat s'anomena: BRerrorRestrNode71.txt. El missatge que apareix per pantalla és:

"Graf no generat. Hi ha un error de validesa.
Comprova el node 71 i les seves restriccions."

5.3.3. Altres errors

En l'anterior apartat, s'ha explicat el problema de validesa del fitxer. També pot haver-hi un error de validesa en el camí. Aquest fet es produeix quan s'intenta seguir un itinerari en que el tren compleix una restricció, és a dir, vol anar enrere.

L'excepció que es llança quan ocorre aquest problema és: **NullPointerException**. Aquesta excepció diu que el valor d'una variable és *null*.

S'explica la manera com s'obté un camí, perquè s'entengui millor això. L'algorisme de *Dijkstra* modificat troba tots els camins de menys cost des de l'origen fins a la resta de nodes en un graf donat. Però l'aplicació, no troba tots els camins. Els busca d'un en un, és a dir, que a més de tenir un origen donat, el destí també s'ha de conèixer.

Es fa servir una variable que s'anomena node precedent. Cada node del graf té aquesta variable. Aleshores, quan s'ha de construir el camí, es comença pel final. Hi ha un recorregut dels nodes precedents en el sentit invers del camí, fins a l'origen. En el cas que s'ha explicat de l'error en la recerca del camí, el problema estava en que ningú era capaç de saber arribar al node X, per tant aquest no podia tenir node precedent.

Ja s'ha esbrinat quina és l'excepció que s'ha de capturar, i en quin moment cal fer-ho. Ara cal afegir un codi per al moment de l'execució. Aquest codi evitarà que s'aturi l'aplicació i que el següent missatge aparegui per pantalla:

"Hi ha un problema amb el camí seleccionat:
no es pot seguir. Comprova-ho!
Fixa't en el node: 'node precedent'"

CAPÍTOL 6. VALORACIÓ

Un cop s'ha acabat la implementació de l'aplicació cal comprovar diversos factors. Primer s'ha de verificar si s'han assolit els objectius que persegueix el projecte. Un cop es comprovi això, s'ha de veure si s'ha respectat la planificació inicial, i si ha variat, de quina manera i per què. Els terminis previstos per assolir els objectius s'han de contrastar amb el desenvolupament del projecte.

6.1. Assoliment dels objectius

Els objectius s'han assolit. L'objectiu principal era crear una aplicació que optimitzés l'encaminament en el transport ferroviari. Això s'ha complert, ja que l'aplicació funciona correctament. A més a més, aquesta aplicació ha de funcionar en un sistema real, concretament a les estacions de tren de Barcelona de la companyia Renfe. Això significa que s'han de fer proves en l'entorn adequat, però aquestes proves tot just s'han iniciat en el moment de redactar aquest treball.

Un altre objectiu era adquirir coneixements de l'arquitectura *Java*, ser capaç de desenvolupar un projecte amb aquest llenguatge de programació. Un cop plantejat un projecte és necessari saber quines eines hi ha disponibles i de quina metodologia es disposa. També s'ha assolit aquest objectiu. S'han après coneixements per cercar la informació necessària de manera eficient.

A mesura que ha avançat el projecte s'han conegut altres mètodes i mecanismes interessants. L'aplicació ha estat dissenyada mitjançant la metodologia UML, i això ha comportat importants avantatges de temps. La interfície gràfica ha suposat un coneixement complementari, ja que no es plantejava com una prioritat al començar el projecte. Si el temps no hagués permès desenvolupar un GUI, l'aplicació només hauria funcionat en mode línia de comandes.

Fer un projecte individualment comporta utilitzar una metodologia en particular. En canvi, quan es desenvolupa un projecte en grup l'esforç es divideix entre les persones que el formen. En un projecte individual ningú fa la feina que no agrada o la més costosa, òbviament tot el projecte el fa una sola persona.
Totes les proves fetes amb l'aplicació han tingut molta importància. Primer, per trobar errors, segon, han servit per trobar noves necessitats i ampliar les prestacions de l'aplicació. Prestacions no imaginades al començar el projecte.

6.2. Variacions respecte la planificació inicial

En relació amb les feines previstes, hi ha una diferència important: el temps destinat a les proves i a la implementació del projecte. Al principi, s'havia pensat trigar menys temps a desenvolupar aquestes dues tasques. A mesura que avançava, es va veure la importància de no retallar temps de proves o d'implementació. Al disseny inicial hi havia unes hores sense assignar, precisament per si alguna feina suposava més temps del previst. Així, el global de temps no va variar significativament.

Les proves previstes en principi no tenien format, és a dir, se sabia que es necessitava un temps per a fer proves, però a mesura que anés avançant la implementació de l'aplicació es decidiria quines proves fer i quan de temps es destinaria.

Respecte a l'aplicació es poden destacar dos elements força importants. El primer és el de la interfície gràfica. Es va preveure utilitzar els paquets d'AWT per a implementar la interfície; de fet, es van fer algunes proves amb aquests paquets. Durant el temps de recerca d'informació, es va trobar un consell de *Sun* de no utilitzar paquets AWT, i utilitzar els paquets *Swing*. Això va fer canviar el rumb de la recerca i de les proves. També és cert, que estava previst estudiar els paquets *Swing* abans de saber que els paquets AWT no eren recomanats.

Un altre canvi en el programa va ser pel mateix motiu. Quan s'estava dissenyant el graf, es buscaven maneres de contenir la informació. Es van trobar dos tipus de contenidors interessants: els vectors i els *arrays*. Els dos tenen elements positius en comú, per tant eren necessàries més dades per escollir-ne un o altre. Després de cercar per la pàgina de *Sun*, aquesta recomanava utilitzar els 'Arraylist', o sigui, cap dels dos que s'havien previst en principi. Això no va ser cap problema, perquè pràcticament tenen els mateixos mètodes, amb determinades excepcions. La classe vector està recomanada en casos que es tinguin processos concurrents, cas que no importa en aquest programa.

6.3. Solució a un cas real?

Com s'ha dit al llarg del projecte, aquest està pensat per a solucionar un cas real. La situació és el sistema de transport ferroviari, on l'element principal és la seguretat. De fet, l'aplicació de l'algorisme modificat només funciona si la seguretat es pot assegurar. És a dir, la capa de seguretat és la que dóna permís al programa per executar-se.
Com que no es pot provar aquesta aplicació immediatament, sense abans haver fet proves *in situ*, és a dir, en el CCT i amb accés a la base de dades que contingui tota la informació de l'estació. És probable que quan s'entregui el treball les proves no estiguin acabades.

La persona encarregada de fer les proves al programa, ha de fer els següents passos:

- Aprendre el funcionament de l'aplicació.
- Trobar tots els camins possibles en una estació.
- Comprovar que tots els camins trobats respectin les restriccions i facin el camí més curt. La correspondència ha de ser total.

Un cop es compleixin els tres passos, sense variar-ne l'ordre i sense deixar-ne cap per fer, l'aplicació estarà en condicions de treballar en aquella estació. És necessari repetir el procediment per a cada estació on es vulgui fer servir. És TOTALMENT indispensable fer la comprovació de manera perfecta, s'ha de recordar que la SEGURETAT és el punt més important.

6.4. Ampliacions futures

Ja s'ha parlat d'algunes millores que s'han fet durant la implementació. Per exemple en la interfície gràfica. Aquesta part del projecte no era una condició necessària al principi.

Dins del codi, hi ha diverses línies que contenen informació sobre el pes de cada enllaç. Una condició imposada en començar, era que cada enllaç tingués el mateix cost. Prendria el valor unitari, 1. D'aquesta manera es dóna prioritat al nombre de circuits de via que ha de travessar un tren per anar de l'origen al destí ja que tots els enllaços tenen el mateix cost.

Si en un futur es volguessin assignar diferents valors a cada enllaç, indicant així rutes millors i rutes pitjors, com si d'una xarxa de comunicacions es tractés. S'haurien d'afegir les dades referents al pes al fitxer que conté la informació del graf. Serien necessaris alguns canvis en el codi, però poca quantitat i senzills.

Si les rutes tinguessin diferent valor en el cost s'estarien afegint restriccions. Tal com el nom del projecte indica, "...algorisme d'encaminament amb múltiples restriccions", l'aplicació implementa un algorisme modificat per acceptar restriccions. A part de la restricció del programa, i de la restricció dels pesos, el programa ha de ser capaç de suportar més ampliacions en aquest sentit.

És per això, que l'aplicació s'ha dissenyat de manera modular, així en un futur, es podran afegir noves utilitats que ara no estan pensades.
Una funció que si que està pensada, i que és útil, és la de dibuixar tant l'estació com el camí que ha de seguir un tren. Una persona que treballa en el CCT necessita tenir informació visual de l'estació. Tanmateix, un cop se sap quina ruta s'ha de seguir, és interessant que es dibuixi a sobre el graf per entendre-ho fàcilment.

Per tant l'opció de dibuixar, que comporta un projecte nou, és una eina molt interessant.

6.5. Conclusions personals

Totes les conclusions a les quals he arribat al final d'aquest projecte penso que són positives.

Per començar perquè el resultat és satisfactori. S'han assolit els objectius dins del temps previst. A més, s'ha pogut anar més enllà en alguns apartats. El projecte està basat en un sistema real, que ja funciona i que s'intenta millorar, això fa que la feina tingui un valor afegit.

He conegut la metodologia UML, que serveix per a dissenyar un model amb precisió i desenvolupar el sistema. És una eina molt interessant que de segur tornaré a utilitzar.

El llarg procés del projecte ha tingut dificultats. Les més importants les he trobat en la implementació de la pràctica, sobretot a l'hora de decidir quines classes de *Sun* feia servir i quines classes necessitava crear. Desenvolupar un projecte sense tenir cap base és una dificultat afegida que jo no havia tingut fins ara. He adquirit els coneixements necessaris per a elaborar un treball de la mateixa magnitud en un futur.

Per a la recerca d'informació he tingut un problema d'excés de dades. Hi ha molta documentació per a crear aplicacions amb *Java*. D'aquesta manera has de triar quina informació és útil i quina no, i a vegades no tens el temps suficient per triar adequadament. Per aquest motiu ha sigut necessari aprendre nous mètodes de recerca més eficients.

Fer aquest projecte m'ha demostrat que només amb esforç no és suficient, també és convenient el fet de pensar, raonar, comparar, decidir, discriminar, sintetitzar, abstraure,... Dedicació és el terme que recull i engloba totes aquestes accions.

CAPÍTOL 7. BIBLIOGRAFIA

[1] Eckel, B., *Piensa en Java*, Ed. Prentice Hall, 2a edició.

[2] Loy, M., Eckestein, R., Wood, D., Elliott, J., Cole, B., *Java Swing*, Ed. O'Reilly and Associates, Inc, 2002. 2nd edition.

[3] Flanagan, D., *Java in a nutshell*, Ed. O'Reilly and Associates, Inc, 1996.

[4] Sun Microsystems (2004 Setembre) *The JFC Swing Tutorial Second Edition* [En línia]. Pàgina web,
URL <http://java.sun.com/docs/books/tutorial/uiswing/examples.html>

[5] Sun Microsystems (2004 setembre) *Java™ 2 Platform, Standard Edition, 1.4.2 API Specification* [En línia].
Pàgina web, URL <http://java.sun.com/j2se/1.4.2/docs/api/>

[6] Programación en castellano (2004, desembre) *Tutorial de Java (APIs de Java, Swing i JFC)* [En línia].
Pàgina web, URL <http://www.programacion.com/tutorial/swing/>

ANNEX 1. DESCRIPCIÓ DE LES CLASSES

En aquest apartat es descriuen detalladament les classes fetes servir en el programa, així com les fetes servir a la interfície. Posteriorment es detallen també el fitxer d'entrada i el fitxer de sortida.

Classe Node

És una de les classes bàsiques de l'aplicació. A l'hora de crear un node, se li han de passar dades per paràmetre. Això és així, perquè només es creen nodes a l'hora de llegir el fitxer, per tant, és una manera d'assegurar-se que s'han llegit bé les dades.

El constructor és així:

public Node(String idnode, int numenllsort, int numenllentr, int numrestr){}

Les dades que necessita el node es passen per paràmetre, i posteriorment a través dels mètodes corresponents.

Després de crear cada node, és necessari afegir-lo al graf. També es treballa amb els nodes en els conjunts que es fan servir a l'algorisme.

Fa servir la classe **Mapejat** i la classe **IdEnll**.

Mètodes:

- *public void setIdNode(String idnode){}*: Aquest mètode agafa la variable *idnode* passada per paràmetre a una variable local.

- *public void setNumEnllSort(int numenllsort){}*: Aquest mètode fa el mateix que l'anterior però amb la variable *numenllsort*, que són el nombre d'enllaços de sortida.

- *public void setNumEnllEntr(int numenllentr) {}*: Actua de la mateixa manera que els dos mètodes anteriors, ho fa amb la variable *numenllentr*.

- *public void setNumEnllRestr(int numrestr) {}*: Últim mètode que actua igual. Aquest cop amb la variable *numrestr*.

- *public void addDistancia(double distancia) {}*: Aquest mètode no es fa servir en el programa. Està pensat per a una ampliació. El pes de cada enllaç és 1 i no cal fer servir la dada. En cas que es fes servir, ho faria de la mateixa manera que els quatre primers mètodes.

- *public void addIdNodePrec(String idnodeprec) {}*: Aquest mètode es fa servir a l'hora d'executar l'algorisme. Cada vegada que un enllaç coneix un node precedent, fa servir aquest mètode.

- *public void addIdEnllSort(IdEnll idsort) {}*: Aquest mètode fa servir la classe **IdEnll**. Un objecte *IdEnll* guarda les dades dels enllaços de sortida d'un node, que són l'identificador i el pes. La dada del pes no es fa servir, però com es veu, està preparada per utilitzar.

- *public void addIdEnllRestr(IdEnll idrestr){}*: Mètode igual a l'anterior, però en aquest cas, cada objecte *IdEnll* creat guardarà les dades dels enllaços restringits. Per saber, si el tren ve d'un sentit, cap a quins nodes no podrà anar.

- *public String getIdNode() {}*: Aquest mètode retorna l'identificador del node.

- *public String getIdNodePrec() {}*: Aquest mètode retorna l'identificador del node precedent. Es fa servir a l'hora de trobar el camí.

- *public int getNumEnllSort() {}*: Aquest mètode retorna el nombre d'enllaços de sortida que té el node. Necessari en diverses parts de l'algorisme.

- *public int getNumEnllEntr() {}*: Aquest mètode retorna el nombre d'enllaços d'entrada del node. Coincideix que els nodes tenen els mateixos enllaços d'entrada i de sortida, ja que totes les arestes es poden recórrer en ambdós sentits. També es considera aquesta dada per a futures ampliacions, per exemple, si es volgués restringir un camí en un sol sentit.

- *public int getNumEnllRestr() {}*: Mètode per a retornar el nombre d'enllaços restringits. Hi pot haver una confusió amb aquesta dada: cada node té una o més restriccions. Vol dir, que si té 3 enllaços, com a molt pot tenir 3 restriccions, o nombre d'enllaços restringits. Ara bé, de cada un d'aquests enllaços, es pot restringir la circulació a més d'un node. El que segur que estarà restringit, és el node del qual prové la restricció.

- *public double getDistancia() {}*: Aquest mètode retorna la distància que hi ha des d'aquest node a l'origen en un camí determinat. Pot variar en un mateix camí quan fem *relax*. Precisament es fa servir en el mètode *relax*.

- *public IdEnll getIdEnllSort(int index) {}*: Aquesta mètode retorna un *IdEnll*, amb informació dels enllaços de sortida. Cadascun dels objectes *IdEnll* del node està guardat en un 'ArrayList', per això se li ha de passar un índex.

- *public IdEnll getIdEnllRestr(int index) {}*: Aquest mètode fa el mateix que l'anterior però amb la informació dels enllaços restringits.

Classe IdEnll

Aquesta classe està ideada per emmagatzemar informació dels enllaços del node, tant dels enllaços d'entrada/sortida com dels enllaços restringits. Com que la informació a desar és poca, s'ha creat una sola classe i no dues.

En el cas que es necessitessin objectes més complexos, es crearien a partir de **IdEnll** amb herència. Posteriorment s'afegirien el que fes falta. Un exemple, seria en el cas que es volguessin afegir restriccions. Hi hauria dues opcions: crear una classe nova, que adquirís les mateixes propietat que **IdEnll** a través de l'herència, i posteriorment afegir-li els elements necessaris. Una altra opció seria modificar la classe **IdEnll** existent per tal d'afegir els nous requeriments.

Es fa servir en la classe **Node**, i a la classe **DijkstraModificat** en la majoria dels seus mètodes.

Mètodes:

- *public void addIdEnllSort(String idenllsort){}*: Aquest mètode afegeix l'identificador de l'enllaç de sortida. Un *IdEnll* pertany a un *Node*, per tant, es coneix de qui és l'enllaç.

- *public void addPes(double pes){}*: Mètode per afegir el pes que té un enllaç, de sortida o d'entrada. Aquest és un més dels mètodes que no es fa servir actualment al projecte. Serviria en cas que els enllaços tinguessin diferent cost.

- *public void addIdEnllRestrProv(String idenllrestrprov){}*: Aquest mètode afegeix l'identificador de l'enllaç d'on prové la restricció a l'*IdEnll* del Node determinat. Es recorda que com a màxim un Node tindrà tants enllaços restringits com enllaços de sortida, i com a mínim en tindrà un.

- *public void addIdEnllRestr(String idrestr){}*: Aquest mètode es pot confondre amb l'anterior. En la classe *Node* ja s'ha explicat la diferència. Per cada node d'on prové restricció, hi pot haver més d'un node al qual no puguem anar. És una matriu lògica. Cada node té sempre una restricció, que és no poder tornar enrere. O sigui, que si es va d' A a B, no es pot anar el següent pas de B a A. A més a més, se suposa que tampoc es pot anar a C. Per tant, si s'està al node B, es tindrà la restricció provinent d' A, i dins d'aquesta restricció, hi haurà dos nodes restringits, que són A i C, als qual es podrà anar quan el sentit sigui A→B.

- *public void addIdEnllRestrArrayList(ArrayList idrestr){}*: Aquest mètode és igual a l'anterior, però afegint directament tot l''ArrayList' de cop. A l'anterior el que es feia era afegir l'identificador d'un en un a l''ArrayList'.

- *public void addNumRestr(int numrestr){}*: Mètode per afegir el nombre d'enllaços dels quals provenen restriccions.

- *public String getIdEnllSort(){}*: Mètode per retornar l'identificador de l'enllaç de sortida.

- *public double getPes(){}*: Mètode per aconseguir el pes de l'enllaç. És un altre mètode que no es fa servir en l'aplicació actual.

- *public String getIdEnllRestrProv(){}*: Aquest mètode retorna l'identificador de l'enllaç del qual prové la restricció.

- *public String getIdRestrSort(int index) {}*: Aquest mètode retorna l'identificador d'un dels enllaços restringits. El retorna en funció de la posició que ocupa en l''Arrayist', índex.

- public int getNumRestr(){}: Aquest mètode retorna el nombre de restriccions que té el Node.

Classe Mapejat

Aquesta classe està pensada per simplificar la feina durant la implementació del programa. Conté informació de tots els identificadors dels nodes del graf. Els ordena dins d'un 'ArrayList', segons l'ordre en què els troba en el fitxer.

Aquesta classe és utilitzada per la classe **Conjunt**, la classe **Graf** i la classe **DijkstraModificat**.

Mètodes:

- *public void addId(String id) {}*: Aquest mètode afegeix els identificadors dels nodes d'un a un. S'emmagatzemen a un 'ArrayList'.

- *public String getIdString(int index) {}*: Mètode per aconseguir un identificador d'un node. Se li passa per paràmetre l'índex de la posició que ocupa en l''ArrayList'.

- *public ArrayList getIdNodes(){}*: Aquest mètode retorna l''ArrayList' que conté tots els identificadors.

- *public String getIdStringL(long i) {}*: Mètode per retornar l'identificador d'un node. Se li passa per paràmetre l'índex de la posició que ocupa en l''ArrayList', però en lloc de passar-li un enter, se li passa un *long*. Això és així, per compatibilitat amb la lectura del fitxer.

- *public int getIdInt(String idnode) {}*: Mètode que retorna la posició d'un node a l''ArrayList' quan se li passa per paràmetre l'identificador.

- *public boolean existeixId(String id){}*: Aquest mètode retorna un *true* si el node existeix en l''ArrayList' i un *false* si no hi és. Se li passa per paràmetre l'identificador del node.

Classe Conjunt

Aquesta classe és necessària per a l'algorisme de *Dijkstra*. Durant els passos de l'algorisme és necessari fer servir dos conjunts, 'Q' i 'S'.

Aquesta classe s'utilitza només a la classe **DijkstraModificat**, que és on es troben els mètodes de l'algorisme.

Fa servir la classe **Mapejat**.

Mètodes:

- *public void iniciArrayList(){}*: Mètode per inicialitzar el conjunt. Esborra tot el que contingui, i posa a zero el valor de nombre de nodes.
- *public void addNode(Node nnode) {}*: Aquest mètode afegeix un node a l''ArrayList'.
- *public void delNode(Node nnode){}*: Mètode per esborrar un node de l''ArrayList'.
- *public ArrayList getArrayListNodes(){}*: Mètode que retorna l''ArrayList'.
- *public void setArrayListNodes(ArrayList denodes){}*: Mètode per afegir l''ArrayList' que conté els nodes del graf.
- *public int esBuit(){}*: Mètode per saber si queden nodes al conjunt.

Classe Graf

Aquesta classe és bàsica, juntament amb la classe **Node** emmagatzema les dades de l'estació.

La seva funció és la de tenir un 'ArrayList' que contingui els nodes i saber quants nodes té.

S'utilitza només a la classe **DijkstraModificat**. Concretament, als mètodes de *llegirFitxer()* i al mètode de l'algorisme.

Fa servir la classe **Mapejat**.

Mètodes:

- *public void setNombreNodes(int nombrenodes){}*: Aquest mètode introdueix el nombre de nodes que té el graf.
- *public void addNode(Node nnode) {}*: Aquest mètode afegeix un node a l''ArrayList'.

- *public void setArrayListNodes(ArrayList denodes){}*: Aquest mètode afegeix un 'ArrayList' de nodes.

- *public int getNombreNodes(){}*: Mètode que retorna el nombre de nodes del graf.

- *public Node getNodeInt(int index) {}*: Mètode que retorna un node si li passem per paràmetre l'índex de la posició que ocupa el node a l''ArrayList'.

- *public Node getNodeString(String id) {}*: Aquest mètode també retorna un node, però ho fa a partir de l'identificador del node.

- *public ArrayList getArrayListNodes() {}*: Mètode que retorna l''ArrayList' de nodes.

Classe *Dijkstra*Modificat

Aquesta classe varia segons la versió de l'aplicació. Bàsicament, les dues versions tenen la mateixa funció. La classe ha de tenir el mètode principal, i per tant, és des d'on s'executa el programa. En el model de línia de comandes aquesta classe té un codi més reduït i més senzill.

El programa amb interfície gràfica té un codi més extens, ja que conté tots els elements que es mostren per pantalla, les seves declaracions, els esdeveniments, etc.

Comparteixen uns mètodes comuns, els referents a l'algorisme. L'aplicació de la interfície gràfica té altres mètodes, la de línia de comandes no. Primer es descriuen els mètodes comuns.

Mètodes:

- *public static Graf llegirFitxer(String nomfitxer) {}*: Aquest mètode és l'encarregat de llegir el fitxer, i guardar les dades que va llegint per tal de formar un graf. El fitxer ha de tenir un format determinat. Retorna el graf que ha format.

- *public static boolean verificarGraf(){}*: Aquest mètode s'encarrega de comprovar la validesa del graf que s'ha creat a partir del fitxer. Verifica dues dades, els enllaços d'entrada/sortida i els enllaços restringits. Retorna un *true* o un *false* segons si la verificació és correcte o no.

- *public static Cami algorisme(Graf ggraf, String idnodeorigen, String idnodedesti){}* : Aquest és el mètode que conté l'algorisme. És el nucli del programa. Per paràmetre necessita que li passin el graf i els circuits de via origen i destí. Retorna el camí que hagi trobat, encara que sigui nul.

- *public static void relax(String idnode1, String idnode2){}*: Mètode que forma part de l'algorisme. És una operació que fa aquest dins d'un bucle, i que repeteix vàries vegades. Modifica dades del graf. Com que *graf* és una variable global, no és necessari passar-lo per paràmetre.

- *public static Node minimaDistancia(Conjunt w){}*: Aquest és un altre mètode de l'algorisme, també conté un bulce. Calcula la mínima distància entre nodes del conjunt 'w', obtingut per paràmetre. Retorna el node amb mínima distància.

- *public static boolean adjacencia(String idnodea, String idnodeb){}*: Mètode que forma part de l'algorisme de *Dijkstra*. Retorna un boleà segons si els dos nodes comparats són adjacents o no. Si ho són retorna un *true*, si no ho són retorna un *false*.

- *public static boolean siRestriccio(String idnodea1, String idnodeb2){}*: Un altre mètode que forma part de l'algorisme. És semblant a l'anterior, però retorna el boleà segons si entre els dos nodes existeix una restricció.

Aquí s'acaben els mètodes compartits, ara es descriuen els mètodes que formen part de la interfície gràfica exclusivament.

Mètodes:

- *private void jBotoGuardaActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode espera un esdeveniment del botó 'Guarda' per executar el seu codi. El seu codi consisteix en obrir la pantalla de guardar fitxers. Guarda en un fitxer l'últim camí trobat.

- *private void jComboDestiActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode està pendent dels esdeveniments del *combo* dels circuits de destí. Si es selecciona un circuit quan ja s'ha generat el graf, es guarda el circuit seleccionat en la variable corresponent.

- *private void jComboOrigenActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode és idèntic a l'anterior però pel circuit de via origen.

- *private void jBotoBuscaActionPerformed(Java.awt.event.ActionEvent evt) {}*: El mètode escolta quan es prem el botó 'Busca'. Si es compleix que ja s'han triat dos circuits de via, i el que això ha comportat, busca el camí entre els dos nodes i l'imprimeix per pantalla.

- *private void jBotoCancelaActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode espera un esdeveniment al botó 'Cancela'. Quan passa això, inicialitza totes les variables i esborra el quadre text. Obliga a l'usuari/a a tornar a començar.

- *private void jBotoGeneraActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode espera l'esdeveniment del botó 'Genera'. Quan es

prem aquest botó s'executa el codi: verifica el graf, i introdueix tots els circuits de via en els *combo*.

- *private void formPropertyChange(Java.beans.PropertyChangeEvent evt) {}*: Aquest mètode canvia les propietats de la finestra principal de la interfície.

- *private void formWindowClosing(Java.awt.event.WindowEvent evt) {}*: El codi d'aquest mètode s'executarà quan algú intenti tancar la finestra principal, sense que importi de quina manera.

- *Private void jDialogInstruccionsPropertyChange (Java.beans.PropertyChangeEvent evt) {}*: Aquest mètode modifica l'aparença del diàleg 'Instruccions'.

- *private void jDialogSobrePropertyChange (Java.beans.PropertyChangeEvent evt) {}*: Aquest mètode actua igual que l'anterior, però modifica l'aparença del diàleg 'Sobre'.

- *private void jDialogSortirPropertyChange (Java.beans.PropertyChangeEvent evt) {}*: Un altre mètode per modificar les característiques d'un diàleg, aquesta vegada del diàleg 'Sortir'.

- *private void jBotoInstruccionsActionPerformed (Java.awt.event.ActionEvent evt) {}*: El codi d'aquest mètode s'executarà quan es tanqui el diàleg de les instruccions. Amaga el diàleg.

- *private void jBotoSobreActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode fa el mateix que l'anterior, però pel diàleg 'Sobre'.

- *private void jBotoEsborraActionPerformed(Java.awt.event.ActionEvent evt) {}*: Mètode que espera l'esdeveniment del botó 'Esborra'. El codi que hi ha serveix per a esborrar la pantalla.

- *private void jPantallaObrirFitxerActionPerformed (Java.awt.event.ActionEvent evt) {}*: Mètode que actua amb la pantalla per a obrir fitxers. Captura el nom i la ruta del fitxer.

- *private void jBotoSiSortirActionPerformed(Java.awt.event.ActionEvent evt) {}*: Mètode que espera l'esdeveniment del botó 'Si' del diàleg 'Sortir'. Si es prem, l'aplicació es tanca.

- *private void jBotoNoSortirActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode és el mateix que l'anterior però del botó 'No'. Si es prem es tanca el diàleg.

- *private void jSobreActionPerformed(Java.awt.event.ActionEvent evt) {}*: Aquest mètode actua quan es prem, dins del menú, l'opció 'Sobre'. El que fa és obrir el diàleg *Sobre*.

- *private void jInstruccionsActionPerformed(Java.awt.event.ActionEvent evt) {}*: Fa el mateix que el mètode anterior, però en aquest cas obre el diàleg *Instruccions*.

- *private void jTancarActionPerformed(Java.awt.event.ActionEvent evt) {}*: Un altre mètode igual. Obre el diàleg *Tancar*.

- *private void jObrirFitxerActionPerformed(Java.awt.event.ActionEvent evt) {}*: Un altre mètode igual. Obre el diàleg *ObriFitxer*.

Aquí s'acaben els mètodes de la interfície gràfica. És necessari controlar cada esdeveniment al qual es vulgui aplicar un codi. Es poden afegir tants mètodes com es vulguin, tenint en compte, que si es fa l'aplicació massa pesada, el temps d'execució augmentarà.

ANNEX 2. DESCRIPCIÓ DEL FITXER D'ENTRADA

A continuació es mostra com és el fitxer que s'ha fet servir al llarg de les proves, i que conté les dades de l'estació de Sants de Barcelona de la companyia Renfe. Es mostra tota la informació des del principi fins al node 9, ja que el fitxer sencer és molt extens i repetitiu, i amb una mostra de 10 nodes n'hi ha prou. El conjunt de quadres de text que hi ha a continuació es mostren un sobre l'altre per tal que s'entengui millor la seva estructura original:

```
/*Nombre de nodes*/
87
CVICV7BR
CVICV8BR
CVICV9BR
CVICV10BR
CVICV11BR
CVICV12BR
CVICV13BR
CVICVR2BR
CVICV1BR
CVICV2BR
CVICV3BR
CVICV6768BR
CVICV4BR
CVICV5BR
CVICV6770BR
CVICV6BR
CVICV40BR
CVICV44BR
CVICV134BR
CVICVFBR
CVICV988BR
CVICVEBR
CVICV3696BR
CVICVR5BR
CVICVA33BR
CVICV976-2BR
CVICV6787-2BR
CVICV6786-2BR
CVICVA977-1BR
CVICV977-1BR
CVICV968-2BR
CVICV6739BR
CVICVA6770BR
CVICV28ABR
CVICV28BBR
CVICV6792-2BR
```

CVICV6797-1BR
CVICV3683-2BR
CVICV6789-1BR
CVICV3692-1BR
CVICVA4BR
CVICVA92BR
CVICVA16BR
CVICVA2BR
CVICVA70BR
CVICVA80BR
CVICVA68BR
CVICVA59BR
CVICVA5BR
CVICVA41BR
CVICVA49BR
CVICVA63BR
CVICVA31BR
CVICVA8BR
CVICVT82BR
CVICVA14BR
CVICVA57BR
CVICVA34BR
CVICVA60BR
CVICVA6BR
CVICVT72BR
CVICVA52BR
CVICVA37BR
CVICVA53BR
CVICVT7BR
CVICVT25BR
CVICVA11BR
CVICVT76BR
CVICVT78BR
CVICVT39BR
CVICVA15BR
CVICV20BR
CVICVR6BR
CVICVR8BR
CVICV3691-1BR
CVICV1003BR
CVICVR3BR
CVICVA74BR
CVICVA43BR
CVICVA21BR
CVICVA44BR
CVICVA27BR
CVICV22BR
CVICV982-1BR
CVICV3678-1BR

```
CVICV3676-2BR
CVICV6800-2BR
/*Node 0*/
2
2
68 1
69 1
/* RESTRICCIONS */
2
68 1 68
69 1 69
/*Node 1*/
2
2
68 1
69 1
/* RESTRICCIONS */
2
68 1 68
69 1 69
/*Node 2*/
2
2
68 1
69 1
/* RESTRICCIONS */
2
68 1 68
69 1 69
/*Node 3*/
2
2
77 1
78 1
/* RESTRICCIONS */
2
77 1 77
78 1 78
/*Node 4*/
2
2
77 1
78 1
/* RESTRICCIONS */
2
77 1 77
78 1 78
/*Node 5*/
2
```

```
2
77 1
78 1
/* RESTRICCIONS */
2
77 1 77
78 1 78
/*Node 6*/
2
2
45 1
51 1
/* RESTRICCIONS */
2
45 1 45
51 1 51
/*Node 7*/
2
2
83 1
40 1
/* RESTRICCIONS */
2
83 1 83
40 1 40
/*Node 8*/
2
2
41 1
56 1
/* RESTRICCIONS */
2
41 1 41
56 1 56
/*Node 9*/
2
2
54 1
62 1
/* RESTRICCIONS */
2
54 1 54
62 1 62
```

Fig.Annex.1 Fitxer d'entrada de l'estació de Sants

ANNEX 3. DESCRIPCIÓ DEL FITXER DE SORTIDA

Ara es mostra el fitxer que es crea quan es troba un camí. Com que apareix el node origen i el node destí, es pot comprovar en el plànol de l'estació que es troba a continuació, que el camí sigui correcte.

```
Node origen:
CVICV977-1BR
Node destí:
CVICV8BR
Camí a seguir:
CVICV977-1BR
CVICV982-1BR
CVICVR2BR
CVICVA4BR
CVICVA8BR
CVICVA14BR
CVICVA34BR
CVICVA44BR
CVICVT72BR
CVICVT78BR
CVICV8BR
```

Fig.Annex.2 Exemple de fitxer de sortida

ANNEX 4. PLÀNOL DE L'ESTACIÓ DE SANTS

En el plànol que es troba a la següent pàgina, es visualitza una estació de tren real. Concretament és l'estació de Sants de Barcelona. És la mateixa imatge que tenen els treballadors/es de la CCT en el seu lloc de treball. Les persones que treballen al centre de control necessiten visualitzar el sistema d'una manera còmode.

El graf està imprès en un full DIN-A3. A la CCT l'estació està formada per dues imatges. Gràcies al programari d'*Adobe* (el *Photoshop*) s'ha aconseguit modificar de tal manera que aparegui la imatge en un sol fitxer. La qualitat del gràfic s'ha millorat, però degut a les proporcions del paper i de la imatge, la resolució no és bona. De totes maneres, es poden llegir els noms dels circuits de via, tot i que amb algunes dificultats.

El full DIN-A3 està orientat horitzontalment i doblegat per la meitat. Es desplega d'esquerra a dreta. Les limitacions físiques que té el format del treball ha obligat ha fer un disseny concret. La impressió és en color.